OpenCV Reference Manual

v2.1

March 18, 2010

Contents

Part I

C API Reference

Chapter 1

cxcore. The Core Functionality

1.1 Basic Structures

CvPoint

2D point with integer coordinates (usually zero-based).

```
typedef struct CvPoint
{
    int x;
    int y;
}
CvPoint;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint cvPoint( int x, int y );

/* Conversion from CvPoint2D32f */
inline CvPoint cvPointFrom32f( CvPoint2D32f point );
```

CvPoint2D32f

2D point with floating-point coordinates

```
typedef struct CvPoint2D32f
{
    float x;
    float y;
}
CvPoint2D32f;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint2D32f cvPoint2D32f( double x, double y );

/* Conversion from CvPoint */
inline CvPoint2D32f cvPointTo32f( CvPoint point );
```

CvPoint3D32f

3D point with floating-point coordinates

```
typedef struct CvPoint3D32f
{
    float x;
    float y;
    float z;
}
CvPoint3D32f;
```

x x-coordinate

y y-coordinate

z z-coordinate

```
/* Constructor */
inline CvPoint3D32f cvPoint3D32f( double x, double y, double z );
```

CvPoint2D64f

2D point with double precision floating-point coordinates

```
typedef struct CvPoint2D64f
{
    double x;
    double y;
}
CvPoint2D64f;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint2D64f cvPoint2D64f( double x, double y );

/* Conversion from CvPoint */
inline CvPoint2D64f cvPointTo64f( CvPoint point );
```

CvPoint3D64f

3D point with double precision floating-point coordinates

```
typedef struct CvPoint3D64f
{
    double x;
    double y;
    double z;
}
CvPoint3D64f;
```

x x-coordinate

y y-coordinate

z z-coordinate

```
/* Constructor */
inline CvPoint3D64f cvPoint3D64f( double x, double y, double z );
```

CvSize

Pixel-accurate size of a rectangle.

```
typedef struct CvSize
{
    int width;
    int height;
}
CvSize;
```

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvSize cvSize( int width, int height );
```

CvSize2D32f

Sub-pixel accurate size of a rectangle.

```
typedef struct CvSize2D32f
{
    float width;
    float height;
}
CvSize2D32f;
```

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvSize2D32f cvSize2D32f( double width, double height );
```

CvRect

Offset (usually the top-left corner) and size of a rectangle.

```
typedef struct CvRect
{
    int x;
    int y;
    int width;
    int height;
}
CvRect;
```

x x-coordinate of the top-left corner

y y-coordinate of the top-left corner (bottom-left for Windows bitmaps)

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvRect cvRect( int x, int y, int width, int height );
```

CvScalar

A container for 1-,2-,3- or 4-tuples of doubles.

```
typedef struct CvScalar
{
    double val[4];
}
CvScalar;
```

```
/* Constructor:
initializes val[0] with val0, val[1] with val1, etc.
*/
inline CvScalar cvScalar( double val0, double val1=0,
                          double val2=0, double val3=0 );
/* Constructor:
initializes all of val[0]...val[3] with val0123
*/
inline CvScalar cvScalarAll( double val0123 );

/* Constructor:
initializes val[0] with val0, and all of val[1]...val[3] with zeros
*/
inline CvScalar cvRealScalar( double val0 );
```

CvTermCriteria

Termination criteria for iterative algorithms.

```
#define CV_TERMCRIT_ITER    1
#define CV_TERMCRIT_NUMBER  CV_TERMCRIT_ITER
#define CV_TERMCRIT_EPS     2

typedef struct CvTermCriteria
{
    int    type;
    int    max_iter;
    double epsilon;
}
CvTermCriteria;
```

type A combination of CV_TERMCRIT_ITER and CV_TERMCRIT_EPS

max_iter Maximum number of iterations

epsilon Required accuracy

```
/* Constructor */
inline CvTermCriteria cvTermCriteria( int type, int max_iter, double epsilon );

/* Check and transform a CvTermCriteria so that
   type=CV_TERMCRIT_ITER+CV_TERMCRIT_EPS
   and both max_iter and epsilon are valid */
CvTermCriteria cvCheckTermCriteria( CvTermCriteria criteria,
                                    double default_eps,
                                    int default_max_iters );
```

CvMat

A multi-channel matrix.

```
typedef struct CvMat
{
    int type;
    int step;

    int* refcount;

    union
```

```
    {
        uchar* ptr;
        short* s;
        int* i;
        float* fl;
        double* db;
    } data;

#ifdef __cplusplus
    union
    {
        int rows;
        int height;
    };

    union
    {
        int cols;
        int width;
    };
#else
    int rows;
    int cols;
#endif

} CvMat;
```

type A CvMat signature (CV_MAT_MAGIC_VAL) containing the type of elements and flags

step Full row length in bytes

refcount Underlying data reference counter

data Pointers to the actual matrix data

rows Number of rows

cols Number of columns

Matrices are stored row by row. All of the rows are aligned by 4 bytes.

CvMatND

Multi-dimensional dense multi-channel array.

```
typedef struct CvMatND
{
    int type;
    int dims;

    int* refcount;

    union
    {
        uchar* ptr;
        short* s;
        int* i;
        float* fl;
        double* db;
    } data;

    struct
    {
        int size;
        int step;
    }
    dim[CV_MAX_DIM];

} CvMatND;
```

type A CvMatND signature (CV_MATND_MAGIC_VAL), combining the type of elements and flags

dims The number of array dimensions

refcount Underlying data reference counter

data Pointers to the actual matrix data

dim For each dimension, the pair (number of elements, distance between elements in bytes)

CvSparseMat

Multi-dimensional sparse multi-channel array.

```
typedef struct CvSparseMat
{
    int type;
    int dims;
    int* refcount;
```

```
    struct CvSet* heap;
    void** hashtable;
    int hashsize;
    int total;
    int valoffset;
    int idxoffset;
    int size[CV_MAX_DIM];

} CvSparseMat;
```

type A CvSparseMat signature (CV_SPARSE_MAT_MAGIC_VAL), combining the type of elements and flags.

dims Number of dimensions

refcount Underlying reference counter. Not used.

heap A pool of hash table nodes

hashtable The hash table. Each entry is a list of nodes.

hashsize Size of the hash table

total Total number of sparse array nodes

valoffset The value offset of the array nodes, in bytes

idxoffset The index offset of the array nodes, in bytes

size Array of dimension sizes

IplImage

IPL image header

```
typedef struct _IplImage
{
    int  nSize;
    int  ID;
    int  nChannels;
    int  alphaChannel;
    int  depth;
    char colorModel[4];
    char channelSeq[4];
```

```
    int  dataOrder;
    int  origin;
    int  align;
    int  width;
    int  height;
    struct _IplROI *roi;
    struct _IplImage *maskROI;
    void  *imageId;
    struct _IplTileInfo *tileInfo;
    int  imageSize;
    char *imageData;
    int  widthStep;
    int  BorderMode[4];
    int  BorderConst[4];
    char *imageDataOrigin;
}
IplImage;
```

nSize `sizeof(IplImage)`

ID Version, always equals 0

nChannels Number of channels. Most OpenCV functions support 1-4 channels.

alphaChannel Ignored by OpenCV

depth Pixel depth in bits. The supported depths are:

- **IPL_DEPTH_8U** Unsigned 8-bit integer
- **IPL_DEPTH_8S** Signed 8-bit integer
- **IPL_DEPTH_16U** Unsigned 16-bit integer
- **IPL_DEPTH_16S** Signed 16-bit integer
- **IPL_DEPTH_32S** Signed 32-bit integer
- **IPL_DEPTH_32F** Single-precision floating point
- **IPL_DEPTH_64F** Double-precision floating point

colorModel Ignored by OpenCV. The OpenCV function CvtColor requires the source and destination color spaces as parameters.

channelSeq Ignored by OpenCV

dataOrder 0 = `IPL_DATA_ORDER_PIXEL` - interleaved color channels, 1 - separate color channels. CreateImage only creates images with interleaved channels. For example, the usual layout of a color image is: $b_{00}g_{00}r_{00}b_{10}g_{10}r_{10}...$

origin 0 - top-left origin, 1 - bottom-left origin (Windows bitmap style)

align Alignment of image rows (4 or 8). OpenCV ignores this and uses widthStep instead.

width Image width in pixels

height Image height in pixels

roi Region Of Interest (ROI). If not NULL, only this image region will be processed.

maskROI Must be NULL in OpenCV

imageId Must be NULL in OpenCV

tileInfo Must be NULL in OpenCV

imageSize Image data size in bytes. For interleaved data, this equals `image->height·image->widthStep`

imageData A pointer to the aligned image data

widthStep The size of an aligned image row, in bytes

BorderMode Border completion mode, ignored by OpenCV

BorderConst Border completion mode, ignored by OpenCV

imageDataOrigin A pointer to the origin of the image data (not necessarily aligned). This is used for image deallocation.

The IplImage structure was inherited from the Intel Image Processing Library, in which the format is native. OpenCV only supports a subset of possible IplImage formats, as outlined in the parameter list above.

In addition to the above restrictions, OpenCV handles ROIs differently. OpenCV functions require that the image size or ROI size of all source and destination images match exactly. On the other hand, the Intel Image Processing Library processes the area of intersection between the source and destination images (or ROIs), allowing them to vary independently.

CvArr

Arbitrary array

```
typedef void CvArr;
```

The metatype `CvArr` is used *only* as a function parameter to specify that the function accepts arrays of multiple types, such as IplImage*, CvMat* or even CvSeq* sometimes. The particular array type is determined at runtime by analyzing the first 4 bytes of the header.

1.2 Operations on Arrays

cvAbsDiff

Calculates absolute difference between two arrays.

```
void cvAbsDiff(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates absolute difference between two arrays.

$$\texttt{dst}(i)_c = |\texttt{src1}(I)_c - \texttt{src2}(I)_c|$$

All the arrays must have the same data type and the same size (or ROI size).

cvAbsDiffS

Calculates absolute difference between an array and a scalar.

```
void cvAbsDiffS(const CvArr* src, CvArr* dst, CvScalar value);
```

```
#define cvAbs(src, dst) cvAbsDiffS(src, dst, cvScalarAll(0))
```

src The source array

dst The destination array

value The scalar

The function calculates absolute difference between an array and a scalar.

$$\texttt{dst}(i)_c = |\texttt{src}(I)_c - \texttt{value}_c|$$

All the arrays must have the same data type and the same size (or ROI size).

cvAdd

Computes the per-element sum of two arrays.

```
void cvAdd(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds one array to another:

```
dst(I)=src1(I)+src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cvAddS

Computes the sum of an array and a scalar.

```
void cvAddS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Added scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds a scalar `value` to every element in the source array `src1` and stores the result in `dst`. For types that have limited range this operation is saturating.

```
dst(I)=src(I)+value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size).

---

cvAddWeighted

Computes the weighted sum of two arrays.

```
void cvAddWeighted(const CvArr* src1, double alpha, const CvArr* src2,
double beta, double gamma, CvArr* dst);
```

src1 The first source array

alpha Weight for the first array elements

src2 The second source array

beta Weight for the second array elements

dst The destination array

gamma Scalar, added to each sum

The function calculates the weighted sum of two arrays as follows:

```
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
```

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvAnd

Calculates per-element bit-wise conjunction of two arrays.

```
void cvAnd(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)&src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cvAndS

Calculates per-element bit-wise conjunction of an array and a scalar.

```
void cvAndS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)&value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

The following sample demonstrates how to calculate the absolute value of floating-point array elements by clearing the most-significant bit:

```
float a[] = { -1, 2, -3, 4, -5, 6, -7, 8, -9 };
CvMat A = cvMat(3, 3, CV\_32F, &a);
int i, absMask = 0x7fffffff;
cvAndS(&A, cvRealScalar(*(float*)&absMask), &A, 0);
for(i = 0; i < 9; i++ )
    printf("%.1f ", a[i]);
```

The code should print:

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

---

cvAvg

Calculates average (mean) of array elements.

```
CvScalar cvAvg(const CvArr* arr, const CvArr* mask=NULL);
```

arr The array

mask The optional operation mask

The function calculates the average value `M` of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$M_c = \frac{\sum_{I, \texttt{mask}(I) \neq 0} \texttt{arr}(I)_c}{N}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average to the first scalar component S_0 .

---

cvAvgSdv

Calculates average (mean) of array elements.

```
void cvAvgSdv(const CvArr* arr, CvScalar* mean, CvScalar* stdDev, const
CvArr* mask=NULL);
```

arr The array

mean Pointer to the output mean value, may be NULL if it is not needed

stdDev Pointer to the output standard deviation

mask The optional operation mask

The function calculates the average value and standard deviation of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$mean_c = \frac{1}{N} \sum_{I, \texttt{mask}(I) \neq 0} \texttt{arr}(I)_c$$
$$stdDev_c = \sqrt{\frac{1}{N} \sum_{I, \texttt{mask}(I) \neq 0} (\texttt{arr}(I)_c - mean_c)^2}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average and standard deviation to the first components of the output scalars ($mean_0$ and $stdDev_0$).

---

cvCalcCovarMatrix

Calculates covariance matrix of a set of vectors.

```
void cvCalcCovarMatrix(
           const CvArr** vects,
           int count,
           CvArr* covMat,
           CvArr* avg,
           int flags);
```

vects The input vectors, all of which must have the same type and the same size. The vectors do not have to be 1D, they can be 2D (e.g., images) and so forth

count The number of input vectors

covMat The output covariance matrix that should be floating-point and square

avg The input or output (depending on the flags) array - the mean (average) vector of the input vectors

flags The operation flags, a combination of the following values

CV_COVAR_SCRAMBLED The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]$$

, that is, the covariance matrix is $\texttt{count} \times \texttt{count}$. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces

technique for face recognition). Eigenvalues of this "scrambled" matrix will match the eigenvalues of the true covariance matrix and the "true" eigenvectors can be easily calculated from the eigenvectors of the "scrambled" covariance matrix.

CV_COVAR_NORMAL The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...] \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T$$

, that is, `covMat` will be a covariance matrix with the same linear size as the total number of elements in each input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

CV_COVAR_USE_AVG If the flag is specified, the function does not calculate `avg` from the input vectors, but, instead, uses the passed `avg` vector. This is useful if `avg` has been already calculated somehow, or if the covariance matrix is calculated by parts - in this case, `avg` is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.

CV_COVAR_SCALE If the flag is specified, the covariance matrix is scaled. In the "normal" mode `scale` is '1./count'; in the "scrambled" mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled ('scale=1').

CV_COVAR_ROWS Means that all the input vectors are stored as rows of a single matrix, `vects[0].count` is ignored in this case, and `avg` should be a single-row vector of an appropriate size.

CV_COVAR_COLS Means that all the input vectors are stored as columns of a single matrix, `vects[0].count` is ignored in this case, and `avg` should be a single-column vector of an appropriate size.

The function calculates the covariance matrix and, optionally, the mean vector of the set of input vectors. The function can be used for PCA, for comparing vectors using Mahalanobis distance and so forth.

---

cvCartToPolar

Calculates the magnitude and/or angle of 2d vectors.

```
void cvCartToPolar(
           const CvArr* x,
           const CvArr* y,
           CvArr* magnitude,
```

```
            CvArr* angle=NULL,
            int angleInDegrees=0);
```

x The array of x-coordinates

y The array of y-coordinates

magnitude The destination array of magnitudes, may be set to NULL if it is not needed

angle The destination array of angles, may be set to NULL if it is not needed. The angles are measured in radians (0 to 2π) or in degrees (0 to 360 degrees).

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

```
magnitude(I)=sqrt(x(I)^2^+y(I)^2^ ),
angle(I)=atan(y(I)/x(I) )
```

The angles are calculated with 0.1 degree accuracy. For the (0,0) point, the angle is set to 0.

---

cvCbrt

Calculates the cubic root

```
float cvCbrt(float value);
```

value The input floating-point value

The function calculates the cubic root of the argument, and normally it is faster than `pow(value,1./3)`. In addition, negative arguments are handled properly. Special values ($\pm\infty$, NaN) are not handled.

---

cvClearND

Clears a specific array element.

```
void cvClearND(CvArr* arr, int* idx);
```

arr Input array

idx Array of the element indices

The function cvClearND clears (sets to zero) a specific element of a dense array or deletes the element of a sparse array. If the sparse array element does not exists, the function does nothing.

---

cvCloneImage

Makes a full copy of an image, including the header, data, and ROI.

```
IplImage* cvCloneImage(const IplImage* image);
```

image The original image

The returned `IplImage*` points to the image copy.

---

cvCloneMat

Creates a full matrix copy.

```
CvMat* cvCloneMat(const CvMat* mat);
```

mat Matrix to be copied

Creates a full copy of a matrix and returns a pointer to the copy.

---

cvCloneMatND

Creates full copy of a multi-dimensional array and returns a pointer to the copy.

```
CvMatND* cvCloneMatND(const CvMatND* mat);
```

mat Input array

---

cvCloneSparseMat

Creates full copy of sparse array.

```
CvSparseMat* cvCloneSparseMat(const CvSparseMat* mat);
```

mat Input array

The function creates a copy of the input array and returns pointer to the copy.

---

cvCmp

Performs per-element comparison of two arrays.

```
void cvCmp(const CvArr* src1, const CvArr* src2, CvArr* dst, int
cmpOp);
```

src1 The first source array

src2 The second source array. Both source arrays must have a single channel.

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

CV_CMP_EQ src1(I) ”equal to” value

CV_CMP_GT src1(I) ”greater than” value

CV_CMP_GE src1(I) ”greater or equal” value

CV_CMP_LT src1(I) ”less than” value

CV_CMP_LE src1(I) ”less or equal” value

CV_CMP_NE src1(I) ”not equal” value

The function compares the corresponding elements of two arrays and fills the destination mask array:

```
dst(I)=src1(I) op src2(I),
```

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

---

cvCmpS

Performs per-element comparison of an array and a scalar.

```
void cvCmpS(const CvArr* src, double value, CvArr* dst, int cmpOp);
```

src The source array, must have a single channel

value The scalar value to compare each array element with

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

CV_CMP_EQ src1(I) ”equal to” value

CV_CMP_GT src1(I) ”greater than” value

CV_CMP_GE src1(I) ”greater or equal” value

CV_CMP_LT src1(I) ”less than” value

CV_CMP_LE src1(I) ”less or equal” value

CV_CMP_NE src1(I) ”not equal” value

The function compares the corresponding elements of an array and a scalar and fills the destination mask array:

```
dst(I)=src(I) op scalar
```

where `op` is $=, >, \geq, <, \leq \ or \ \neq$.

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same size (or ROI size).

cvConvertScale

Converts one array to another with optional linear transformation.

```
void cvConvertScale(const CvArr* src, CvArr* dst, double scale=1,
double shift=0);
```

```
#define cvCvtScale cvConvertScale
#define cvScale  cvConvertScale
#define cvConvert(src, dst )  cvConvertScale((src), (dst), 1, 0 )
```

src Source array

dst Destination array

scale Scale factor

shift Value added to the scaled source array elements

The function has several different purposes, and thus has several different names. It copies one array to another with optional scaling, which is performed first, and/or optional type conversion, performed after:

$$\texttt{dst}(I) = \texttt{scale}\texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)$$

All the channels of multi-channel arrays are processed independently.

The type of conversion is done with rounding and saturation, that is if the result of scaling + conversion can not be represented exactly by a value of the destination array element type, it is set to the nearest representable value on the real axis.

In the case of `scale=1, shift=0` no prescaling is done. This is a specially optimized case and it has the appropriate cvConvert name. If source and destination array types have equal types, this is also a special case that can be used to scale and shift a matrix or an image and that is caled cvScale.

cvConvertScaleAbs

Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

```
void cvConvertScaleAbs(const CvArr* src, CvArr* dst, double scale=1,
double shift=0);
```

```
#define cvCvtScaleAbs cvConvertScaleAbs
```

src Source array

dst Destination array (should have 8u depth)

scale ScaleAbs factor

shift Value added to the scaled source array elements

The function is similar to cvConvertScale, but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scale}\texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of cvConvertScale and cvAbs functions.

---

cvCopy

Copies one array to another.

```
void cvCopy(const CvArr* src, CvArr* dst, const CvArr* mask=NULL);
```

src The source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies selected elements from an input array to an output array:

$$\texttt{dst}(I) = \texttt{src}(I) \quad \text{if} \quad \texttt{mask}(I) \neq 0.$$

If any of the passed arrays is of `IplImage` type, then its ROI and COI fields are used. Both arrays must have the same type, the same number of dimensions, and the same size. The function can also copy sparse arrays (mask is not supported in this case).

cvCountNonZero

Counts non-zero array elements.

```
int cvCountNonZero(const CvArr* arr);
```

arr The array must be a single-channel array or a multi-channel image with COI set

The function returns the number of non-zero elements in arr:

$$\sum_I (\texttt{arr}(I) \neq 0)$$

In the case of `IplImage` both ROI and COI are supported.

cvCreateData

Allocates array data

```
void cvCreateData(CvArr* arr);
```

arr Array header

The function allocates image, matrix or multi-dimensional array data. Note that in the case of matrix types OpenCV allocation functions are used and in the case of IplImage they are used unless `CV_TURN_ON_IPL_COMPATIBILITY` was called. In the latter case IPL functions are used to allocate the data.

cvCreateImage

Creates an image header and allocates the image data.

```
IplImage* cvCreateImage(CvSize size, int depth, int channels);
```

size Image width and height

depth Bit depth of image elements. See IplImage for valid depths.

channels Number of channels per pixel. See IplImage for details. This function only creates images with interleaved channels.

This call is a shortened form of

```
header = cvCreateImageHeader(size, depth, channels);
cvCreateData(header);
```

---

cvCreateImageHeader

Creates an image header but does not allocate the image data.

```
IplImage* cvCreateImageHeader(CvSize size, int depth, int channels);
```

size Image width and height

depth Image depth (see cvCreateImage)

channels Number of channels (see cvCreateImage)

This call is an analogue of

```
hdr=iplCreateImageHeader(channels, 0, depth,
                         channels == 1 ? "GRAY" : "RGB",
                         channels == 1 ? "GRAY" : channels == 3 ? "BGR" :
                         channels == 4 ? "BGRA" : "",
                         IPL_DATA_ORDER_PIXEL, IPL_ORIGIN_TL, 4,
                         size.width, size.height,
                         0,0,0,0);
```

but it does not use IPL functions by default (see the `CV_TURN_ON_IPL_COMPATIBILITY` macro).

---

cvCreateMat

Creates a matrix header and allocates the matrix data.

```
CvMat* cvCreateMat(
           int rows,
           int cols,
           int type);
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type The type of the matrix elements in the form `CV_<bit depth><S|U|F>C<number of channels>`, where S=signed, U=unsigned, F=float. For example, CV_8UC1 means the elements are 8-bit unsigned and the there is 1 channel, and CV_32SC2 means the elements are 32-bit signed and there are 2 channels.

This is the concise form for:

```
CvMat* mat = cvCreateMatHeader(rows, cols, type);
cvCreateData(mat);
```

---

cvCreateMatHeader

Creates a matrix header but does not allocate the matrix data.

```
CvMat* cvCreateMatHeader(
           int rows,
           int cols,
           int type);
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements, see cvCreateMat

The function allocates a new matrix header and returns a pointer to it. The matrix data can then be allocated using cvCreateData or set explicitly to user-allocated data via cvSetData.

---

cvCreateMatND

Creates the header and allocates the data for a multi-dimensional dense array.

```
CvMatND* cvCreateMatND(
           int dims,
           const int* sizes,
           int type);
```

dims Number of array dimensions. This must not exceed CV_MAX_DIM (32 by default, but can be changed at build time).

sizes Array of dimension sizes.

type Type of array elements, see cvCreateMat.

This is a short form for:

```
CvMatND* mat = cvCreateMatNDHeader(dims, sizes, type);
cvCreateData(mat);
```

---

cvCreateMatNDHeader

Creates a new matrix header but does not allocate the matrix data.

```
CvMatND* cvCreateMatNDHeader(
            int dims,
            const int* sizes,
            int type);
```

dims Number of array dimensions

sizes Array of dimension sizes

type Type of array elements, see cvCreateMat

The function allocates a header for a multi-dimensional dense array. The array data can further be allocated using cvCreateData or set explicitly to user-allocated data via cvSetData.

---

cvCreateSparseMat

Creates sparse array.

```
CvSparseMat* cvCreateSparseMat(int dims, const int* sizes, int type);
```

dims Number of array dimensions. In contrast to the dense matrix, the number of dimensions is practically unlimited (up to 2^{16}).

sizes Array of dimension sizes

type Type of array elements. The same as for CvMat

The function allocates a multi-dimensional sparse array. Initially the array contain no elements, that is cvGet or cvGetReal returns zero for every index.

---

cvCrossProduct

Calculates the cross product of two 3D vectors.

```
void cvCrossProduct(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source vector

src2 The second source vector

dst The destination vector

The function calculates the cross product of two 3D vectors:

$$\texttt{dst} = \texttt{src1} \times \texttt{src2}$$

or:

$$\begin{aligned}\texttt{dst}_1 &= \texttt{src1}_2\texttt{src2}_3 - \texttt{src1}_3\texttt{src2}_2 \\ \texttt{dst}_2 &= \texttt{src1}_3\texttt{src2}_1 - \texttt{src1}_1\texttt{src2}_3 \\ \texttt{dst}_3 &= \texttt{src1}_1\texttt{src2}_2 - \texttt{src1}_2\texttt{src2}_1\end{aligned}$$

---

cvDCT

Performs a forward or inverse Discrete Cosine transform of a 1D or 2D floating-point array.

```
void cvDCT(const CvArr* src, CvArr* dst, int flags);
```

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_ROWS     4
```

src Source array, real 1D or 2D array

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

CV_DXT_FORWARD do a forward 1D or 2D transform.

CV_DXT_INVERSE do an inverse 1D or 2D transform.

CV_DXT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

The function performs a forward or inverse transform of a 1D or 2D floating-point array:
Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C^{(N)}_{jk} = \sqrt{\alpha_j / N} \cos\left(\frac{\pi(2k+1)j}{2N}\right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.
Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)}\right)^{-1} \cdot Y = \left(C^{(N)}\right)^{T} \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)}\right)^{T} = I$)
Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)}\right)^{T}$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)}\right)^{T} \cdot X \cdot C^{(N)}$$

---

cvDFT

Performs a forward or inverse Discrete Fourier transform of a 1D or 2D floating-point array.

```
void cvDFT(const CvArr* src, CvArr* dst, int flags, int nonzeroRows=0);
```

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_SCALE    2
#define CV_DXT_ROWS     4
#define CV_DXT_INV_SCALE (CV_DXT_SCALE|CV_DXT_INVERSE)
#define CV_DXT_INVERSE_SCALE CV_DXT_INV_SCALE
```

src Source array, real or complex

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

CV_DXT_FORWARD do a forward 1D or 2D transform. The result is not scaled.

CV_DXT_INVERSE do an inverse 1D or 2D transform. The result is not scaled. CV_DXT_FORWARD and CV_DXT_INVERSE are mutually exclusive, of course.

CV_DXT_SCALE scale the result: divide it by the number of array elements. Usually, it is combined with CV_DXT_INVERSE, and one may use a shortcut CV_DXT_INV_SCALE.

CV_DXT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows the user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

nonzeroRows Number of nonzero rows in the source array (in the case of a forward 2d transform), or a number of rows of interest in the destination array (in the case of an inverse 2d transform). If the value is negative, zero, or greater than the total number of rows, it is ignored. The parameter can be used to speed up 2d convolution/correlation when computing via DFT. See the example below.

The function performs a forward or inverse transform of a 1D or 2D floating-point array:
Forward Fourier transform of 1D vector of N elements:

$$y = F^{(N)} \cdot x, where F^{(N)}_{jk} = exp(-i \cdot 2\pi \cdot j \cdot k/N)$$

,

$$i = sqrt(-1)$$

Inverse Fourier transform of 1D vector of N elements:

$$x' = (F^{(N)})^{-1} \cdot y = conj(F^(N)) \cdot yx = (1/N) \cdot x$$

Forward Fourier transform of 2D vector of M $\times$ N elements:

$$Y = F^{(M)} \cdot X \cdot F^{(N)}$$

Inverse Fourier transform of 2D vector of M $\times$ N elements:

$$X' = conj(F^{(M)}) \cdot Y \cdot conj(F^{(N)})X = (1/(M \cdot N)) \cdot X'$$

In the case of real (single-channel) data, the packed format, borrowed from IPL, is used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix} ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\ ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\ ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\ \hdots & & & & & & & & \\ ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \hdots & & & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\ ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \hdots & & & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\ ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \hdots & & & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2} \end{bmatrix}$$

Note: the last column is present if `N` is even, the last row is present if `M` is even. In the case of 1D real transform the result looks like the first row of the above matrix.

Here is the example of how to compute 2D convolution using DFT.

```
CvMat* A = cvCreateMat(M1, N1, CVg32F);
CvMat* B = cvCreateMat(M2, N2, A->type);

// it is also possible to have only abs(M2-M1)+1 times abs(N2-N1)+1
// part of the full convolution result
CvMat* conv = cvCreateMat(A->rows + B->rows - 1, A->cols + B->cols - 1,
                          A->type);

// initialize A and B
...

int dftgM = cvGetOptimalDFTSize(A->rows + B->rows - 1);
int dftgN = cvGetOptimalDFTSize(A->cols + B->cols - 1);

CvMat* dftgA = cvCreateMat(dft\_M, dft\_N, A->type);
CvMat* dftgB = cvCreateMat(dft\_M, dft\_N, B->type);
```

```
CvMat tmp;

// copy A to dftgA and pad dft\_A with zeros
cvGetSubRect(dftgA, &tmp, cvRect(0,0,A->cols,A->rows));
cvCopy(A, &tmp);
cvGetSubRect(dftgA, &tmp, cvRect(A->cols,0,dft\_A->cols - A->cols,A->rows));
cvZero(&tmp);
// no need to pad bottom part of dftgA with zeros because of
// use nonzerogrows parameter in cvDFT() call below

cvDFT(dftgA, dft\_A, CV\_DXT\_FORWARD, A->rows);

// repeat the same with the second array
cvGetSubRect(dftgB, &tmp, cvRect(0,0,B->cols,B->rows));
cvCopy(B, &tmp);
cvGetSubRect(dftgB, &tmp, cvRect(B->cols,0,dft\_B->cols - B->cols,B->rows));
cvZero(&tmp);
// no need to pad bottom part of dftgB with zeros because of
// use nonzerogrows parameter in cvDFT() call below

cvDFT(dftgB, dft\_B, CV\_DXT\_FORWARD, B->rows);

cvMulSpectrums(dftgA, dft\_B, dft\_A, 0 /* or CV\_DXT\_MUL\_CONJ to get
                correlation rather than convolution */);

cvDFT(dftgA, dft\_A, CV\_DXT\_INV\_SCALE, conv->rows); // calculate only
                                                      // the top part
cvGetSubRect(dftgA, &tmp, cvRect(0,0,conv->cols,conv->rows));

cvCopy(&tmp, conv);
```

---

cvDecRefData

Decrements an array data reference counter.

```
void cvDecRefData(CvArr* arr);
```

arr Pointer to an array header

The function decrements the data reference counter in a CvMat or CvMatND if the reference counter pointer is not NULL. If the counter reaches zero, the data is deallocated. In the current

implementation the reference counter is not NULL only if the data was allocated using the cvCreateData function. The counter will be NULL in other cases such as: external data was assigned to the header using cvSetData, the matrix header is part of a larger matrix or image, or the header was converted from an image or n-dimensional matrix header.

---

cvDet

Returns the determinant of a matrix.

```
double cvDet(const CvArr* mat);
```

mat The source matrix

The function returns the determinant of the square matrix `mat`. The direct method is used for small matrices and Gaussian elimination is used for larger matrices. For symmetric positive-determined matrices, it is also possible to run cvSVD with $U = V = 0$ and then calculate the determinant as a product of the diagonal elements of W.

---

cvDiv

Performs per-element division of two arrays.

```
void cvDiv(const CvArr* src1, const CvArr* src2, CvArr* dst, double
scale=1);
```

src1 The first source array. If the pointer is NULL, the array is assumed to be all 1's.

src2 The second source array

dst The destination array

scale Optional scale factor

The function divides one array by another:

$$\texttt{dst}(I) = \begin{cases} \texttt{scale} \cdot \texttt{src1}(I)/\texttt{src2}(I) & \text{if } \texttt{src1} \text{ is not } \texttt{NULL} \\ \texttt{scale}/\texttt{src2}(I) & \text{otherwise} \end{cases}$$

All the arrays must have the same type and the same size (or ROI size).

cvDotProduct

Calculates the dot product of two arrays in Euclidian metrics.

```
double cvDotProduct(const CvArr* src1, const CvArr* src2);
```

src1 The first source array

src2 The second source array

The function calculates and returns the Euclidean dot product of two arrays.

$$src1 \bullet src2 = \sum_I (\texttt{src1}(I)\texttt{src2}(I))$$

In the case of multiple channel arrays, the results for all channels are accumulated. In particular, `cvDotProduct(a,a)` where `a` is a complex vector, will return $||\texttt{a}||^2$. The function can process multi-dimensional arrays, row by row, layer by layer, and so on.

cvEigenVV

Computes eigenvalues and eigenvectors of a symmetric matrix.

```
void cvEigenVV(
           CvArr* mat,
           CvArr* evects,
           CvArr* evals,
           double eps=0,
           int lowindex = 0,
           int highindex = 0);
```

mat The input symmetric square matrix, modified during the processing

evects The output matrix of eigenvectors, stored as subsequent rows

evals The output vector of eigenvalues, stored in the descending order (order of eigenvalues and eigenvectors is syncronized, of course)

eps Accuracy of diagonalization. Typically, DBL_EPSILON (about 10^{-15}) works well. THIS PARAMETER IS CURRENTLY IGNORED.

lowindex Optional index of largest eigenvalue/-vector to calculate. (See below.)

highindex Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The function computes the eigenvalues and eigenvectors of matrix A:

```
mat*evects(i,:)' = evals(i)*evects(i,:)' (in MATLAB notation)
```

If either low- or highindex is supplied the other is required, too. Indexing is 1-based. Example: To calculate the largest eigenvector/-value set lowindex = highindex = 1. For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

The contents of matrix A is destroyed by the function.

Currently the function is slower than cvSVD yet less accurate, so if A is known to be positively-defined (for example, it is a covariance matrix)it is recommended to use cvSVD to find eigenvalues and eigenvectors of A, especially if eigenvectors are not required.

---

cvExp

Calculates the exponent of every array element.

```
void cvExp(const CvArr* src, CvArr* dst);
```

src The source array

dst The destination array, it should have double type or the same type as the source

The function calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}(I)}$$

The maximum relative error is about 7×10^{-6}. Currently, the function converts denormalized values to zeros on output.

cvFastArctan

Calculates the angle of a 2D vector.

```
float cvFastArctan(float y, float x);
```

x x-coordinate of 2D vector

y y-coordinate of 2D vector

The function calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0 degrees to 360 degrees. The accuracy is about 0.1 degrees.

cvFlip

Flip a 2D array around vertical, horizontal or both axes.

```
void cvFlip(const CvArr* src, CvArr* dst=NULL, int flipMode=0);
```

```
#define cvMirror cvFlip
```

src Source array

dst Destination array. If $\texttt{dst} = \texttt{NULL}$ the flipping is done in place.

flipMode Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas:

The function flips the array in one of three different ways (row and column indices are 0-based):

$$dst(i,j) = \begin{cases} \texttt{src}(rows(\texttt{src}) - i - 1, j) & \text{if } \texttt{flipMode} = 0 \\ \texttt{src}(i, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} > 0 \\ \texttt{src}(rows(\texttt{src}) - i - 1, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (flipMode = 0) to switch between top-left and bottom-left image origin, which is a typical operation in video processing under Win32 systems.

- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (flipMode > 0)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flipMode < 0)
- reversing the order of 1d point arrays (flipMode ¿ 0)

---

cvGEMM

Performs generalized matrix multiplication.

```
void cvGEMM(
           const CvArr* src1,
           const CvArr* src2, double alpha,
           const CvArr* src3,
           double beta,
           CvArr* dst,
           int tABC=0);
#define cvMatMulAdd(src1, src2, src3, dst ) cvGEMM(src1, src2, 1, src3,
1, dst, 0 )
           #define cvMatMul(src1, src2, dst ) cvMatMulAdd(src1, src2,
0, dst )
```

src1 The first source array

src2 The second source array

src3 The third source array (shift). Can be NULL, if there is no shift.

dst The destination array

tABC The operation flags that can be 0 or a combination of the following values

CV_GEMM_A_T transpose src1

CV_GEMM_B_T transpose src2

CV_GEMM_C_T transpose src3

For example, CV_GEMM_A_T+CV_GEMM_C_T corresponds to

$$\texttt{alpha}\,\texttt{src1}^T\,\texttt{src2} + \texttt{beta}\,\texttt{src3}^T$$

The function performs generalized matrix multiplication:

$$\texttt{dst} = \texttt{alpha}\, op(\texttt{src1})\, op(\texttt{src2}) + \texttt{beta}\, op(\texttt{src3}) \quad \text{where } op(X) \text{ is } X \text{ or } X^T$$

All the matrices should have the same data type and coordinated sizes. Real or complex floating-point matrices are supported.

---

cvGet?D

Return a specific array element.

```
CvScalar cvGet1D(const CvArr* arr, int idx0); CvScalar cvGet2D(const
CvArr* arr, int idx0, int idx1); CvScalar cvGet3D(const CvArr* arr,
int idx0, int idx1, int idx2); CvScalar cvGetND(const CvArr* arr, int*
idx);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

The functions return a specific array element. In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

---

cvGetCol(s)

Returns array column or column span.

```
CvMat* cvGetCol(const CvArr* arr, CvMat* submat, int col);
```

```
CvMat* cvGetCols(const CvArr* arr, CvMat* submat, int startCol, int
endCol);
```

arr Input array

submat Pointer to the resulting sub-array header

col Zero-based index of the selected column

startCol Zero-based index of the starting column (inclusive) of the span

endCol Zero-based index of the ending column (exclusive) of the span

The functions `GetCol` and `GetCols` return the header, corresponding to a specified column span of the input array. `GetCol` is a shortcut for cvGetCols:

```
cvGetCol(arr, submat, col); // ~ cvGetCols(arr, submat, col, col + 1);
```

---

cvGetDiag

Returns one of array diagonals.

```
CvMat* cvGetDiag(const CvArr* arr, CvMat* submat, int diag=0);
```

arr Input array

submat Pointer to the resulting sub-array header

diag Array diagonal. Zero corresponds to the main diagonal, -1 corresponds to the diagonal above the main , 1 corresponds to the diagonal below the main, and so forth.

The function returns the header, corresponding to a specified diagonal of the input array.

---

cvGetDims, cvGetDimSize

Return number of array dimensions and their sizes or the size of a particular dimension.

```
int cvGetDims(const CvArr* arr, int* sizes=NULL);
```

```
int cvGetDimSize(const CvArr* arr, int index);
```

arr Input array

sizes Optional output vector of the array dimension sizes. For 2d arrays the number of rows (height) goes first, number of columns (width) next.

index Zero-based dimension index (for matrices 0 means number of rows, 1 means number of columns; for images 0 means height, 1 means width)

The function `cvGetDims` returns the array dimensionality and the array of dimension sizes. In the case of `IplImage` or CvMat it always returns 2 regardless of number of image/matrix rows. The function `cvGetDimSize` returns the particular dimension size (number of elements per that dimension). For example, the following code calculates total number of array elements in two ways:

```
// via cvGetDims()
int sizes[CV_MAX_DIM];
int i, total = 1;
int dims = cvGetDims(arr, size);
for(i = 0; i < dims; i++ )
    total *= sizes[i];

// via cvGetDims() and cvGetDimSize()
int i, total = 1;
int dims = cvGetDims(arr);
for(i = 0; i < dims; i++ )
    total *= cvGetDimsSize(arr, i);
```

---

cvGetElemType

Returns type of array elements.

```
int cvGetElemType(const CvArr* arr);
```

arr Input array

The function returns type of the array elements as described in cvCreateMat discussion: `CV_8UC1` ... `CV_64FC4`.

cvGetImage

Returns image header for arbitrary array.

```
IplImage* cvGetImage(const CvArr* arr, IplImage* imageHeader);
```

arr Input array

imageHeader Pointer to `IplImage` structure used as a temporary buffer

The function returns the image header for the input array that can be a matrix - CvMat , or an image - `IplImage*`. In the case of an image the function simply returns the input pointer. In the case of CvMat it initializes an `imageHeader` structure with the parameters of the input matrix. Note that if we transform `IplImage` to CvMat and then transform CvMat back to IplImage, we can get different headers if the ROI is set, and thus some IPL functions that calculate image stride from its width and align may fail on the resultant image.

cvGetImageCOI

Returns the index of the channel of interest.

```
int cvGetImageCOI(const IplImage* image);
```

image A pointer to the image header

Returns the channel of interest of in an IplImage. Returned values correspond to the `coi` in cvSetImageCOI.

cvGetImageROI

Returns the image ROI.

```
CvRect cvGetImageROI(const IplImage* image);
```

image A pointer to the image header

If there is no ROI set, `cvRect(0,0,image->width,image->height)` is returned.

cvGetMat

Returns matrix header for arbitrary array.

```
CvMat* cvGetMat(const CvArr* arr, CvMat* header, int* coi=NULL, int
allowND=0);
```

arr Input array

header Pointer to CvMat structure used as a temporary buffer

coi Optional output parameter for storing COI

allowND If non-zero, the function accepts multi-dimensional dense arrays (CvMatND*) and returns 2D (if CvMatND has two dimensions) or 1D matrix (when CvMatND has 1 dimension or more than 2 dimensions). The array must be continuous.

The function returns a matrix header for the input array that can be a matrix - CvMat , an image - `IplImage` or a multi-dimensional dense array - CvMatND (latter case is allowed only if `allowND != 0`) . In the case of matrix the function simply returns the input pointer. In the case of `IplImage*` or CvMatND it initializes the `header` structure with parameters of the current image ROI and returns the pointer to this temporary structure. Because COI is not supported by CvMat , it is returned separately.

The function provides an easy way to handle both types of arrays - `IplImage` and CvMat - using the same code. Reverse transform from CvMat to `IplImage` can be done using the cvGetImage function.

Input array must have underlying data allocated or attached, otherwise the function fails.

If the input array is `IplImage` with planar data layout and COI set, the function returns the pointer to the selected plane and COI = 0. It enables per-plane processing of multi-channel images with planar data layout using OpenCV functions.

cvGetNextSparseNode

Returns the next sparse matrix element

```
CvSparseNode* cvGetNextSparseNode(CvSparseMatIterator* matIterator);
```

matIterator Sparse array iterator

The function moves iterator to the next sparse matrix element and returns pointer to it. In the current version there is no any particular order of the elements, because they are stored in the hash table. The sample below demonstrates how to iterate through the sparse matrix:

Using cvInitSparseMatIterator and cvGetNextSparseNode to calculate sum of floating-point sparse array.

```
double sum;
int i, dims = cvGetDims(array);
CvSparseMatIterator mat_iterator;
CvSparseNode* node = cvInitSparseMatIterator(array, &mat_iterator);

for(; node != 0; node = cvGetNextSparseNode(&mat_iterator ))
{
    /* get pointer to the element indices */
    int* idx = CV_NODE_IDX(array, node);
    /* get value of the element (assume that the type is CV_32FC1) */
    float val = *(float*)CV_NODE_VAL(array, node);
    printf("(");
    for(i = 0; i < dims; i++ )
        printf("%4d%s", idx[i], i < dims - 1 "," : "): ");
    printf("%g\n", val);

    sum += val;
}

printf("\nTotal sum = %g\n", sum);
```

---

cvGetOptimalDFTSize

Returns optimal DFT size for a given vector size.

```
int cvGetOptimalDFTSize(int size0);
```

size0 Vector size

The function returns the minimum number N that is greater than or equal to size0, such that the DFT of a vector of size N can be computed fast. In the current implementation $N = 2^p \times 3^q \times 5^r$, for some p, q, r.

The function returns a negative number if size0 is too large (very close to INT_MAX)

cvGetRawData

Retrieves low-level information about the array.

```
void cvGetRawData(const CvArr* arr, uchar** data, int* step=NULL,
CvSize* roiSize=NULL);
```

arr Array header

data Output pointer to the whole image origin or ROI origin if ROI is set

step Output full row length in bytes

roiSize Output ROI size

The function fills output variables with low-level information about the array data. All output parameters are optional, so some of the pointers may be set to `NULL`. If the array is `IplImage` with ROI set, the parameters of ROI are returned.

The following example shows how to get access to array elements. GetRawData calculates the absolute value of the elements in a single-channel, floating-point array.

```
float* data;
int step;

CvSize size;
int x, y;

cvGetRawData(array, (uchar**)&data, &step, &size);
step /= sizeof(data[0]);

for(y = 0; y < size.height; y++, data += step )
    for(x = 0; x < size.width; x++ )
        data[x] = (float)fabs(data[x]);
```

cvGetReal?D

Return a specific element of single-channel array.

```
double cvGetReal1D(const CvArr* arr, int idx0);
double cvGetReal2D(const CvArr* arr, int idx0, int idx1);
double cvGetReal3D(const CvArr* arr, int idx0, int idx1, int idx2);
double cvGetRealND(const CvArr* arr, int* idx);
```

arr Input array. Must have a single channel.

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

The functions `cvGetReal*D` return a specific element of a single-channel array. If the array has multiple channels, a runtime error is raised. Note that cvGet function can be used safely for both single-channel and multiple-channel arrays though they are a bit slower.

In the case of a sparse array the functions return 0 if the requested node does not exist (no new node is created by the functions).

---

cvGetRow(s)

Returns array row or row span.

```
CvMat* cvGetRow(const CvArr* arr, CvMat* submat, int row);
```

```
CvMat* cvGetRows(const CvArr* arr, CvMat* submat, int startRow, int
endRow, int deltaRow=1);
```

arr Input array

submat Pointer to the resulting sub-array header

row Zero-based index of the selected row

startRow Zero-based index of the starting row (inclusive) of the span

endRow Zero-based index of the ending row (exclusive) of the span

deltaRow Index step in the row span. That is, the function extracts every `deltaRow`-th row from `startRow` and up to (but not including) `endRow`.

The functions return the header, corresponding to a specified row/row span of the input array. Note that `GetRow` is a shortcut for cvGetRows:

```
cvGetRow(arr, submat, row ) ~ cvGetRows(arr, submat, row, row + 1, 1);
```

---

cvGetSize

Returns size of matrix or image ROI.

```
CvSize cvGetSize(const CvArr* arr);
```

arr array header

The function returns number of rows (CvSize::height) and number of columns (CvSize::width) of the input matrix or image. In the case of image the size of ROI is returned.

---

cvGetSubRect

Returns matrix header corresponding to the rectangular sub-array of input image or matrix.

```
CvMat* cvGetSubRect(const CvArr* arr, CvMat* submat, CvRect rect);
```

arr Input array

submat Pointer to the resultant sub-array header

rect Zero-based coordinates of the rectangle of interest

The function returns header, corresponding to a specified rectangle of the input array. In other words, it allows the user to treat a rectangular part of input array as a stand-alone array. ROI is taken into account by the function so the sub-array of ROI is actually extracted.

cvInRange

Checks that array elements lie between the elements of two other arrays.

```
void cvInRange(const CvArr* src, const CvArr* lower, const CvArr*
upper, CvArr* dst);
```

src The first source array

lower The inclusive lower boundary array

upper The exclusive upper boundary array

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0 \wedge \texttt{lower}(I)_1 <= \texttt{src}(I)_1 < \texttt{upper}(I)_1$$

For two-channel arrays and so forth,
dst(I) is set to 0xff (all 1-bits) if src(I) is within the range and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size).

cvInRangeS

Checks that array elements lie between two scalars.

```
void cvInRangeS(const CvArr* src, CvScalar lower, CvScalar upper,
CvArr* dst);
```

src The first source array

lower The inclusive lower boundary

upper The exclusive upper boundary

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0 \wedge \texttt{lower}_1 <= \texttt{src}(I)_1 < \texttt{upper}_1$$

For two-channel arrays nd so forth,
'dst(I)' is set to 0xff (all 1-bits) if 'src(I)' is within the range and 0 otherwise. All the arrays must have the same size (or ROI size).

---

cvIncRefData

Increments array data reference counter.

```
int cvIncRefData(CvArr* arr);
```

arr Array header

The function increments CvMat or CvMatND data reference counter and returns the new counter value if the reference counter pointer is not NULL, otherwise it returns zero.

---

cvInitImageHeader

Initializes an image header that was previously allocated.

```
IplImage* cvInitImageHeader(
           IplImage* image,
           CvSize size,
           int depth,
           int channels,
           int origin=0,
           int align=4);
```

image Image header to initialize

size Image width and height

depth Image depth (see cvCreateImage)

channels Number of channels (see cvCreateImage)

origin Top-left `IPL_ORIGIN_TL` or bottom-left `IPL_ORIGIN_BL`

align Alignment for image rows, typically 4 or 8 bytes

The returned `IplImage*` points to the initialized header.

---

cvInitMatHeader

Initializes a pre-allocated matrix header.

```
CvMat* cvInitMatHeader(
            CvMat* mat,
            int rows,
            int cols,
            int type,
            void* data=NULL,
            int step=CV_AUTOSTEP);
```

mat A pointer to the matrix header to be initialized

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements, see cvCreateMat.

data Optional: data pointer assigned to the matrix header

step Optional: full row width in bytes of the assigned data. By default, the minimal possible step is used which assumes there are no gaps between subsequent rows of the matrix.

This function is often used to process raw data with OpenCV matrix functions. For example, the following code computes the matrix product of two matrices, stored as ordinary arrays:

```
double a[] = { 1, 2, 3, 4,
               5, 6, 7, 8,
               9, 10, 11, 12 };

double b[] = { 1, 5, 9,
               2, 6, 10,
               3, 7, 11,
               4, 8, 12 };

double c[9];
CvMat Ma, Mb, Mc ;

cvInitMatHeader(&Ma, 3, 4, CV_64FC1, a);
cvInitMatHeader(&Mb, 4, 3, CV_64FC1, b);
cvInitMatHeader(&Mc, 3, 3, CV_64FC1, c);

cvMatMulAdd(&Ma, &Mb, 0, &Mc);
// the c array now contains the product of a (3x4) and b (4x3)
```

---

cvInitMatNDHeader

Initializes a pre-allocated multi-dimensional array header.

```
CvMatND* cvInitMatNDHeader(
            CvMatND* mat,
            int dims,
            const int* sizes,
            int type,
            void* data=NULL);
```

mat A pointer to the array header to be initialized

dims The number of array dimensions

sizes An array of dimension sizes

type Type of array elements, see cvCreateMat

data Optional data pointer assigned to the matrix header

cvInitSparseMatIterator

Initializes sparse array elements iterator.

```
CvSparseNode* cvInitSparseMatIterator(const CvSparseMat* mat,
CvSparseMatIterator* matIterator);
```

mat Input array

matIterator Initialized iterator

The function initializes iterator of sparse array elements and returns pointer to the first element, or NULL if the array is empty.

cvInvSqrt

Calculates the inverse square root.

```
float cvInvSqrt(float value);
```

value The input floating-point value

The function calculates the inverse square root of the argument, and normally it is faster than `1./sqrt(value)`. If the argument is zero or negative, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

cvInvert

Finds the inverse or pseudo-inverse of a matrix.

```
double cvInvert(const CvArr* src, CvArr* dst, int method=CV_LU);
```

```
#define cvInv cvInvert
```

src The source matrix

dst The destination matrix

method Inversion method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix

The function inverts matrix `src1` and stores the result in `src2`.

In the case of `LU` method, the function returns the `src1` determinant (src1 must be square). If it is 0, the matrix is not inverted and `src2` is filled with zeros.

In the case of `SVD` methods, the function returns the inversed condition of `src1` (ratio of the smallest singular value to the largest singular value) and 0 if `src1` is all zeros. The SVD methods calculate a pseudo-inverse matrix if `src1` is singular.

---

cvIsInf

Determines if the argument is Infinity.

```
int cvIsInf(double value);
```

value The input floating-point value

The function returns 1 if the argument is $\pm\infty$ (as defined by IEEE754 standard), 0 otherwise.

---

cvIsNaN

Determines if the argument is Not A Number.

```
int cvIsNaN(double value);
```

value The input floating-point value

The function returns 1 if the argument is Not A Number (as defined by IEEE754 standard), 0 otherwise.

cvLUT

Performs a look-up table transform of an array.

```
void cvLUT(const CvArr* src, CvArr* dst, const CvArr* lut);
```

src Source array of 8-bit elements

dst Destination array of a given depth and of the same number of channels as the source array

lut Look-up table of 256 elements; should have the same depth as the destination array. In the case of multi-channel source and destination arrays, the table should either have a single-channel (in this case the same table is used for all channels) or the same number of channels as the source/destination array.

The function fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of `src` as follows:

$$\texttt{dst}_i \leftarrow \texttt{lut}_{\texttt{src}_i + d}$$

where

$$d = \begin{cases} 0 & \text{if } \texttt{src} \text{ has depth } \texttt{CV\_8U} \\ 128 & \text{if } \texttt{src} \text{ has depth } \texttt{CV\_8S} \end{cases}$$

cvLog

Calculates the natural logarithm of every array element's absolute value.

```
void cvLog(const CvArr* src, CvArr* dst);
```

src The source array

dst The destination array, it should have `double` type or the same type as the source

The function calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}[I] = \begin{cases} \log|\texttt{src}(I) & \text{if } \texttt{src}[I] \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where `C` is a large negative number (about -700 in the current implementation).

cvMahalonobis

Calculates the Mahalonobis distance between two vectors.

```
double cvMahalanobis(
            const CvArr* vec1,
            const CvArr* vec2,
            CvArr* mat);
```

vec1 The first 1D source vector

vec2 The second 1D source vector

mat The inverse covariance matrix

The function calculates and returns the weighted distance between two vectors:

$$d(\texttt{vec1},\texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the cvCalcCovarMatrix function and further inverted using the cvInvert function (CV_SVD method is the prefered one because the matrix might be singular).

cvMat

Initializes matrix header (lightweight variant).

```
CvMat cvMat(
            int rows,
            int cols,
            int type,
            void* data=NULL);
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements - see cvCreateMat

data Optional data pointer assigned to the matrix header

Initializes a matrix header and assigns data to it. The matrix is filled *row*-wise (the first `cols` elements of data form the first row of the matrix, etc.)

This function is a fast inline substitution for cvInitMatHeader. Namely, it is equivalent to:

```
CvMat mat;
cvInitMatHeader(&mat, rows, cols, type, data, CV\_AUTOSTEP);
```

---

cvMax

Finds per-element maximum of two arrays.

```
void cvMax(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cvMaxS

Finds per-element maximum of array and scalar.

```
void cvMaxS(const CvArr* src, double value, CvArr* dst);
```

src The first source array

value The scalar value

dst The destination array

The function calculates per-element maximum of array and scalar:

$$\texttt{dst}(I) = \max(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cvMerge

Composes a multi-channel array from several single-channel arrays or inserts a single channel into the array.

```
void cvMerge(const CvArr* src0, const CvArr* src1, const CvArr* src2,
const CvArr* src3, CvArr* dst);
```

```
#define cvCvtPlaneToPix cvMerge
```

src0 Input channel 0

src1 Input channel 1

src2 Input channel 2

src3 Input channel 3

dst Destination array

The function is the opposite to cvSplit. If the destination array has N channels then if the first N input channels are not NULL, they all are copied to the destination array; if only a single source channel of the first N is not NULL, this particular channel is copied into the destination array; otherwise an error is raised. The rest of the source channels (beyond the first N) must always be NULL. For IplImage cvCopy with COI set can be also used to insert a single channel into the image.

---

cvMin

Finds per-element minimum of two arrays.

```
void cvMin(const CvArr* src1, const CvArr* src2, CvArr* dst);
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cvMinMaxLoc

Finds global minimum and maximum in array or subarray.

```
void cvMinMaxLoc(const CvArr* arr, double* minVal, double* maxVal,
CvPoint* minLoc=NULL, CvPoint* maxLoc=NULL, const CvArr* mask=NULL);
```

arr The source array, single-channel or multi-channel with COI set

minVal Pointer to returned minimum value

maxVal Pointer to returned maximum value

minLoc Pointer to returned minimum location

maxLoc Pointer to returned maximum location

mask The optional mask used to select a subarray

The function finds minimum and maximum element values and their positions. The extremums are searched across the whole array, selected `ROI` (in the case of `IplImage`) or, if `mask` is not `NULL`, in the specified array region. If the array has more than one channel, it must be `IplImage` with `COI` set. In the case of multi-dimensional arrays, `minLoc->x` and `maxLoc->x` will contain raw (linear) positions of the extremums.

cvMinS

Finds per-element minimum of an array and a scalar.

```
void cvMinS(const CvArr* src, double value, CvArr* dst);
```

src The first source array

value The scalar value

dst The destination array

The function calculates minimum of an array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

cvMixChannels

Copies several channels from input arrays to certain channels of output arrays

```
void cvMixChannels(const CvArr** src, int srcCount,
           CvArr** dst, int dstCount,
           const int* fromTo, int pairCount);
```

src Input arrays

srcCount The number of input arrays.

dst Destination arrays

dstCount The number of output arrays.

fromTo The array of pairs of indices of the planes copied. `fromTo[k*2]` is the 0-based index of the input channel in `src` and `fromTo[k*2+1]` is the index of the output channel in `dst`. Here the continuous channel numbering is used, that is, the first input image channels are indexed from `0` to `channels(src[0])-1`, the second input image channels are indexed from `channels(src[0])` to `channels(src[0]) + channels(src[1])-1` etc., and the same scheme is used for the output image channels. As a special case, when `fromTo[k*2]` is negative, the corresponding output channel is filled with zero.

The function is a generalized form of cvcvSplit and cvMerge and some forms of CvtColor . It can be used to change the order of the planes, add/remove alpha channel, extract or insert a single plane or multiple planes etc.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B swapped) and separate alpha channel image:

```
CvMat* rgba = cvCreateMat(100, 100, CV_8UC4);
CvMat* bgr = cvCreateMat(rgba->rows, rgba->cols, CV_8UC3);
CvMat* alpha = cvCreateMat(rgba->rows, rgba->cols, CV_8UC1);
cvSet(rgba, cvScalar(1,2,3,4));

CvArr* out[] = { bgr, alpha };
int from_to[] = { 0,2,  1,1,  2,0,  3,3 };
cvMixChannels(&bgra, 1, out, 2, from_to, 4);
```

---

cvMul

Calculates the per-element product of two arrays.

```
void cvMul(const CvArr* src1, const CvArr* src2, CvArr* dst, double
scale=1);
```

src1 The first source array

src2 The second source array

dst The destination array

scale Optional scale factor

The function calculates the per-element product of two arrays:

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I)$$

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

cvMulSpectrums

Performs per-element multiplication of two Fourier spectrums.

```
void cvMulSpectrums(
           const CvArr* src1,
           const CvArr* src2,
           CvArr* dst,
           int flags);
```

src1 The first source array

src2 The second source array

dst The destination array of the same type and the same size as the source arrays

flags A combination of the following values;

- **CV_DXT_ROWS** treats each row of the arrays as a separate spectrum (see cvDFT parameters description).
- **CV_DXT_MUL_CONJ** conjugate the second source array before the multiplication.

The function performs per-element multiplication of the two CCS-packed or complex matrices that are results of a real or complex Fourier transform.

The function, together with cvDFT, may be used to calculate convolution of two arrays rapidly.

cvMulTransposed

Calculates the product of an array and a transposed array.

```
void cvMulTransposed(const CvArr* src, CvArr* dst, int order, const
CvArr* delta=NULL, double scale=1.0);
```

src The source matrix

dst The destination matrix. Must be `CV_32F` or `CV_64F`.

order Order of multipliers

delta An optional array, subtracted from src before multiplication

scale An optional scaling

The function calculates the product of src and its transposition:

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})(\texttt{src} - \texttt{delta})^T$$

if $\texttt{order} = 0$, and

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})^T(\texttt{src} - \texttt{delta})$$

otherwise.

---

cvNorm

Calculates absolute array norm, absolute difference norm, or relative difference norm.

```
double cvNorm(const CvArr* arr1, const CvArr* arr2=NULL, int
normType=CV_L2, const CvArr* mask=NULL);
```

arr1 The first source image

arr2 The second source image. If it is NULL, the absolute norm of arr1 is calculated, otherwise the absolute or relative norm of arr1-arr2 is calculated.

normType Type of norm, see the discussion

mask The optional operation mask

The function calculates the absolute norm of arr1 if arr2 is NULL:

$$norm = \begin{cases} ||\texttt{arr1}||_C = \max_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV\_C} \\ ||\texttt{arr1}||_{L1} = \sum_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV\_L1} \\ ||\texttt{arr1}||_{L2} = \sqrt{\sum_I \texttt{arr1}(I)^2} & \text{if } \texttt{normType} = \texttt{CV\_L2} \end{cases}$$

or the absolute difference norm if arr2 is not NULL:

$$norm = \begin{cases} ||\texttt{arr1} - \texttt{arr2}||_C = \max_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV\_C} \\ ||\texttt{arr1} - \texttt{arr2}||_{L1} = \sum_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV\_L1} \\ ||\texttt{arr1} - \texttt{arr2}||_{L2} = \sqrt{\sum_I (\texttt{arr1}(I) - \texttt{arr2}(I))^2} & \text{if } \texttt{normType} = \texttt{CV\_L2} \end{cases}$$

or the relative difference norm if `arr2` is not NULL and `(normType & CV_RELATIVE) != 0`:

$$norm = \begin{cases} \frac{||\texttt{arr1}-\texttt{arr2}||_C}{||\texttt{arr2}||_C} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_C} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L1}}{||\texttt{arr2}||_{L1}} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_L1} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L2}}{||\texttt{arr2}||_{L2}} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_L2} \end{cases}$$

The function returns the calculated norm. A multiple-channel array is treated as a single-channel, that is, the results for all channels are combined.

---

cvNot

Performs per-element bit-wise inversion of array elements.

```
void cvNot(const CvArr* src, CvArr* dst);
```

src The source array

dst The destination array

The function Not inverses every bit of every array element:

```
dst(I)=~src(I)
```

---

cvOr

Calculates per-element bit-wise disjunction of two arrays.

```
void cvOr(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise disjunction of two arrays:

```
dst(I)=src1(I)|src2(I)
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cvOrS

Calculates a per-element bit-wise disjunction of an array and a scalar.

```
void cvOrS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function OrS calculates per-element bit-wise disjunction of an array and a scalar:

```
dst(I)=src(I)|value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cvPerspectiveTransform

Performs perspective matrix transformation of a vector array.

```
void cvPerspectiveTransform(const CvArr* src, CvArr* dst, const CvMat*
mat);
```

src The source three-channel floating-point array

dst The destination three-channel floating-point array

mat 3×3 or 4×4 transformation matrix

The function transforms every element of `src` (by treating it as 2D or 3D vector) in the following way:

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

---

cvPolarToCart

Calculates Cartesian coordinates of 2d vectors represented in polar form.

```
void cvPolarToCart(
           const CvArr* magnitude,
           const CvArr* angle,
           CvArr* x,
           CvArr* y,
           int angleInDegrees=0);
```

magnitude The array of magnitudes. If it is NULL, the magnitudes are assumed to be all 1's.

angle The array of angles, whether in radians or degrees

x The destination array of x-coordinates, may be set to NULL if it is not needed

y The destination array of y-coordinates, mau be set to NULL if it is not needed

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the x-coodinate, y-coordinate or both of every vector `magnitude(I)*exp(ang j=sqrt(-1)`:

```
x(I)=magnitude(I)*cos(angle(I)),
y(I)=magnitude(I)*sin(angle(I))
```

---

cvPow

Raises every array element to a power.

```
void cvPow(
           const CvArr* src,
           CvArr* dst,
           double power);
```

src The source array

dst The destination array, should be the same type as the source

power The exponent of power

The function raises every element of the input array to `p`:

$$\texttt{dst}[I] = \begin{cases} \texttt{src}(I)^p & \text{if } \texttt{p} \text{ is integer} \\ |\texttt{src}(I)^p| & \text{otherwise} \end{cases}$$

That is, for a non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following example, computing the cube root of array elements, shows:

```
CvSize size = cvGetSize(src);
CvMat* mask = cvCreateMat(size.height, size.width, CVg8UC1);
cvCmpS(src, 0, mask, CVgCMPgLT); /* find negative elements */
cvPow(src, dst, 1./3);
cvSubRS(dst, cvScalarAll(0), dst, mask); /* negate the results of negative inputs */
cvReleaseMat(&mask);
```

For some values of `power`, such as integer values, 0.5, and -0.5, specialized faster algorithms are used.

cvPtr?D

Return pointer to a particular array element.

```
uchar* cvPtr1D(const CvArr* arr, int idx0, int* type=NULL);
uchar* cvPtr2D(const CvArr* arr, int idx0, int idx1, int* type=NULL);
uchar* cvPtr3D(const CvArr* arr, int idx0, int idx1, int idx2, int*
type=NULL);
uchar* cvPtrND(const CvArr* arr, int* idx, int* type=NULL, int
createNode=1, unsigned* precalcHashval=NULL);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

type Optional output parameter: type of matrix elements

createNode Optional input parameter for sparse matrices. Non-zero value of the parameter means that the requested element is created if it does not exist already.

precalcHashval Optional input parameter for sparse matrices. If the pointer is not NULL, the function does not recalculate the node hash value, but takes it from the specified location. It is useful for speeding up pair-wise operations (TODO: provide an example)

The functions return a pointer to a specific array element. Number of array dimension should match to the number of indices passed to the function except for `cvPtr1D` function that can be used for sequential access to 1D, 2D or nD dense arrays.

The functions can be used for sparse arrays as well - if the requested node does not exist they create it and set it to zero.

All these as well as other functions accessing array elements (cvGet, cvGetReal, cvSet, cvSetReal) raise an error in case if the element index is out of range.

cvRNG

Initializes a random number generator state.

```
CvRNG cvRNG(int64 seed=-1);
```

seed 64-bit value used to initiate a random sequence

The function initializes a random number generator and returns the state. The pointer to the state can be then passed to the cvRandInt, cvRandReal and cvRandArr functions. In the current implementation a multiply-with-carry generator is used.

cvRandArr

Fills an array with random numbers and updates the RNG state.

```
void cvRandArr(
           CvRNG* rng,
           CvArr* arr,
           int distType,
           CvScalar param1,
           CvScalar param2);
```

rng RNG state initialized by cvRNG

arr The destination array

distType Distribution type

CV_RAND_UNI uniform distribution

CV_RAND_NORMAL normal or Gaussian distribution

param1 The first parameter of the distribution. In the case of a uniform distribution it is the inclusive lower boundary of the random numbers range. In the case of a normal distribution it is the mean value of the random numbers.

param2 The second parameter of the distribution. In the case of a uniform distribution it is the exclusive upper boundary of the random numbers range. In the case of a normal distribution it is the standard deviation of the random numbers.

The function fills the destination array with uniformly or normally distributed random numbers.

In the example below, the function is used to add a few normally distributed floating-point numbers to random locations within a 2d array.

```
/* let noisy_screen be the floating-point 2d array that is to be "crapped" */
CvRNG rng_state = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* allocate the array of coordinates of points */
CvMat* locations = cvCreateMat(pointCount, 1, CV_32SC2);
/* arr of random point values */
CvMat* values = cvCreateMat(pointCount, 1, CV_32FC1);
CvSize size = cvGetSize(noisy_screen);

/* initialize the locations */
cvRandArr(&rng_state, locations, CV_RAND_UNI, cvScalar(0,0,0,0),
          cvScalar(size.width,size.height,0,0));

/* generate values */
cvRandArr(&rng_state, values, CV_RAND_NORMAL,
          cvRealScalar(100), // average intensity
          cvRealScalar(30) // deviation of the intensity
         );

/* set the points */
for(i = 0; i < pointCount; i++ )
{
    CvPoint pt = *(CvPoint*)cvPtr1D(locations, i, 0);
    float value = *(float*)cvPtr1D(values, i, 0);
    *((float*)cvPtr2D(noisy_screen, pt.y, pt.x, 0 )) += value;
}

/* not to forget to release the temporary arrays */
cvReleaseMat(&locations);
cvReleaseMat(&values);

/* RNG state does not need to be deallocated */
```

---

cvRandInt

Returns a 32-bit unsigned integer and updates RNG.

```
unsigned cvRandInt(CvRNG* rng);
```

rng RNG state initialized by `RandInit` and, optionally, customized by `RandSetRange` (though, the latter function does not affect the discussed function outcome)

The function returns a uniformly-distributed random 32-bit unsigned integer and updates the RNG state. It is similar to the rand() function from the C runtime library, but it always generates a 32-bit number whereas rand() returns a number in between 0 and `RAND_MAX` which is 2^{16} or 2^{32}, depending on the platform.

The function is useful for generating scalar random numbers, such as points, patch sizes, table indices, etc., where integer numbers of a certain range can be generated using a modulo operation and floating-point numbers can be generated by scaling from 0 to 1 or any other specific range.

Here is the example from the previous function discussion rewritten using cvRandInt:

```
/* the input and the task is the same as in the previous sample. */
CvRNG rnggstate = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* ... - no arrays are allocated here */
CvSize size = cvGetSize(noisygscreen);
/* make a buffer for normally distributed numbers to reduce call overhead */
#define bufferSize 16
float normalValueBuffer[bufferSize];
CvMat normalValueMat = cvMat(bufferSize, 1, CVg32F, normalValueBuffer);
int valuesLeft = 0;

for(i = 0; i < pointCount; i++ )
{
    CvPoint pt;
    /* generate random point */
    pt.x = cvRandInt(&rnggstate ) % size.width;
    pt.y = cvRandInt(&rnggstate ) % size.height;

    if(valuesLeft <= 0 )
    {
        /* fulfill the buffer with normally distributed numbers
           if the buffer is empty */
        cvRandArr(&rnggstate, &normalValueMat, CV\_RAND\_NORMAL,
                    cvRealScalar(100), cvRealScalar(30));
        valuesLeft = bufferSize;
    }
    *((float*)cvPtr2D(noisygscreen, pt.y, pt.x, 0 ) =
                                  normalValueBuffer[--valuesLeft];
}

/* there is no need to deallocate normalValueMat because we have
both the matrix header and the data on stack. It is a common and efficient
practice of working with small, fixed-size matrices */
```

cvRandReal

Returns a floating-point random number and updates RNG.

```
double cvRandReal(CvRNG* rng);
```

rng RNG state initialized by cvRNG

The function returns a uniformly-distributed random floating-point number between 0 and 1 (1 is not included).

cvReduce

Reduces a matrix to a vector.

```
void cvReduce(const CvArr* src, CvArr* dst, int dim = -1, int
op=CV_REDUCE_SUM);
```

src The input matrix.

dst The output single-row/single-column vector that accumulates somehow all the matrix rows/-columns.

dim The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row, 1 means that the matrix is reduced to a single column and -1 means that the dimension is chosen automatically by analysing the dst size.

op The reduction operation. It can take of the following values:

- **CV_REDUCE_SUM** The output is the sum of all of the matrix's rows/columns.
- **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
- **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/-columns.
- **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/-columns.

The function reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In the case of CV_REDUCE_SUM and CV_REDUCE_AVG the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

---

cvReleaseData

Releases array data.

```
void cvReleaseData(CvArr* arr);
```

arr Array header

The function releases the array data. In the case of CvMat or CvMatND it simply calls cvDecRefData(), that is the function can not deallocate external data. See also the note to cvCreateData.

---

cvReleaseImage

Deallocates the image header and the image data.

```
void cvReleaseImage(IplImage** image);
```

image Double pointer to the image header

This call is a shortened form of

```
if(*image )
{
    cvReleaseData(*image);
    cvReleaseImageHeader(image);
}
```

cvReleaseImageHeader

Deallocates an image header.

```
void cvReleaseImageHeader(IplImage** image);
```

image Double pointer to the image header

This call is an analogue of

```
if(image )
{
    iplDeallocate(*image, IPL_IMAGE_HEADER | IPL_IMAGE_ROI);
    *image = 0;
}
```

but it does not use IPL functions by default (see the CV_TURN_ON_IPL_COMPATIBILITY macro).

cvReleaseMat

Deallocates a matrix.

```
void cvReleaseMat(CvMat** mat);
```

mat Double pointer to the matrix

The function decrements the matrix data reference counter and deallocates matrix header. If the data reference counter is 0, it also deallocates the data.

```
if(*mat )
    cvDecRefData(*mat);
cvFree((void**)mat);
```

cvReleaseMatND

Deallocates a multi-dimensional array.

```
void cvReleaseMatND(CvMatND** mat);
```

mat Double pointer to the array

The function decrements the array data reference counter and releases the array header. If the reference counter reaches 0, it also deallocates the data.

```
if(*mat )
    cvDecRefData(*mat);
cvFree((void**)mat);
```

---

cvReleaseSparseMat

Deallocates sparse array.

```
void cvReleaseSparseMat(CvSparseMat** mat);
```

mat Double pointer to the array

The function releases the sparse array and clears the array pointer upon exit.

---

cvRepeat

Fill the destination array with repeated copies of the source array.

```
void cvRepeat(const CvArr* src, CvArr* dst);
```

src Source array, image or matrix

dst Destination array, image or matrix

The function fills the destination array with repeated copies of the source array:

```
dst(i,j)=src(i mod rows(src), j mod cols(src))
```

So the destination array may be as larger as well as smaller than the source array.

cvResetImageROI

Resets the image ROI to include the entire image and releases the ROI structure.

```
void cvResetImageROI(IplImage* image);
```

image A pointer to the image header

This produces a similar result to the following, but in addition it releases the ROI structure.

```
cvSetImageROI(image, cvRect(0, 0, image->width, image->height ));
cvSetImageCOI(image, 0);
```

cvReshape

Changes shape of matrix/image without copying data.

```
CvMat* cvReshape(const CvArr* arr, CvMat* header, int newCn, int
newRows=0);
```

arr Input array

header Output header to be filled

newCn New number of channels. 'newCn = 0' means that the number of channels remains unchanged.

newRows New number of rows. 'newRows = 0' means that the number of rows remains unchanged unless it needs to be changed according to `newCn` value.

The function initializes the CvMat header so that it points to the same data as the original array but has a different shape - different number of channels, different number of rows, or both.

The following example code creates one image buffer and two image headers, the first is for a 320x240x3 image and the second is for a 960x240x1 image:

```
IplImage* color_img = cvCreateImage(cvSize(320,240), IPL_DEPTH_8U, 3);
CvMat gray_mat_hdr;
IplImage gray_img_hdr, *gray_img;
cvReshape(color_img, &gray_mat_hdr, 1);
gray_img = cvGetImage(&gray_mat_hdr, &gray_img_hdr);
```

And the next example converts a 3x3 matrix to a single 1x9 vector:

```
CvMat* mat = cvCreateMat(3, 3, CV_32F);
CvMat row_header, *row;
row = cvReshape(mat, &row_header, 0, 1);
```

---

cvReshapeMatND

Changes the shape of a multi-dimensional array without copying the data.

```
CvArr* cvReshapeMatND(const CvArr* arr, int sizeofHeader, CvArr*
header, int newCn, int newDims, int* newSizes);
```

```
#define cvReshapeND(arr, header, newCn, newDims, newSizes )    \
      cvReshapeMatND((arr), sizeof(*(header)), (header),          \
                     (newCn), (newDims), (newSizes))
```

arr Input array

sizeofHeader Size of output header to distinguish between IplImage, CvMat and CvMatND output headers

header Output header to be filled

newCn New number of channels. `newCn` $= 0$ means that the number of channels remains unchanged.

newDims New number of dimensions. `newDims` $= 0$ means that the number of dimensions remains the same.

newSizes Array of new dimension sizes. Only `newDims` $- 1$ values are used, because the total number of elements must remain the same. Thus, if `newDims` $= 1$, `newSizes` array is not used.

The function is an advanced version of cvReshape that can work with multi-dimensional arrays as well (though it can work with ordinary images and matrices) and change the number of dimensions.

Below are the two samples from the cvReshape description rewritten using cvReshapeMatND:

```
IplImage* color_img = cvCreateImage(cvSize(320,240), IPL_DEPTH_8U, 3);
IplImage gray_img_hdr, *gray_img;
gray_img = (IplImage*)cvReshapeND(color_img, &gray_img_hdr, 1, 0, 0);

...

/* second example is modified to convert 2x2x2 array to 8x1 vector */
int size[] = { 2, 2, 2 };
CvMatND* mat = cvCreateMatND(3, size, CV_32F);
CvMat row_header, *row;
row = (CvMat*)cvReshapeND(mat, &row_header, 0, 1, 0);
```

---

cvRound, cvFloor, cvCeil

Converts a floating-point number to an integer.

```
int cvRound(double value); int cvFloor(double value); int cvCeil(double
value);
```

value The input floating-point value

The functions convert the input floating-point number to an integer using one of the rounding modes. `Round` returns the nearest integer value to the argument. `Floor` returns the maximum integer value that is not larger than the argument. `Ceil` returns the minimum integer value that is not smaller than the argument. On some architectures the functions work much faster than the standard cast operations in C. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

---

cvScaleAdd

Calculates the sum of a scaled array and another array.

```
void cvScaleAdd(const CvArr* src1, CvScalar scale, const CvArr* src2,
CvArr* dst);
```

src1 The first source array

scale Scale factor for the first array

src2 The second source array

dst The destination array

```
#define cvMulAddS cvScaleAdd
```

The function calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale}\,\texttt{src1}(I) + \texttt{src2}(I)$$

All array parameters should have the same type and the same size.

---

cvSet

Sets every element of an array to a given value.

```
void cvSet(CvArr* arr, CvScalar value, const CvArr* mask=NULL);
```

arr The destination array

value Fill value

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies the scalar `value` to every selected element of the destination array:

$$\texttt{arr}(I) = \texttt{value} \quad \text{if} \quad \texttt{mask}(I) \neq 0$$

If array `arr` is of `IplImage` type, then is ROI used, but COI must not be set.

---

cvSet?D

Change the particular array element.

```
void cvSet1D(CvArr* arr, int idx0, CvScalar value);
void cvSet2D(CvArr* arr, int idx0, int idx1, CvScalar value);
void cvSet3D(CvArr* arr, int idx0, int idx1, int idx2, CvScalar value);
void cvSetND(CvArr* arr, int* idx, CvScalar value);
```

arr Input array

idx0 The first zero-based component of the element index

idx1 The second zero-based component of the element index

idx2 The third zero-based component of the element index

idx Array of the element indices

value The assigned value

The functions assign the new value to a particular array element. In the case of a sparse array the functions create the node if it does not exist yet.

---

cvSetData

Assigns user data to the array header.

```
void cvSetData(CvArr* arr, void* data, int step);
```

arr Array header

data User data

step Full row length in bytes

The function assigns user data to the array header. Header should be initialized before using `cvCreate*Header`, `cvInit*Header` or cvMat (in the case of matrix) function.

---

cvSetIdentity

Initializes a scaled identity matrix.

```
void cvSetIdentity(CvArr* mat, CvScalar value=cvRealScalar(1));
```

mat The matrix to initialize (not necesserily square)

value The value to assign to the diagonal elements

The function initializes a scaled identity matrix:

$$\texttt{arr}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

---

cvSetImageCOI

Sets the channel of interest in an IplImage.

```
void cvSetImageCOI(
           IplImage* image,
           int coi);
```

image A pointer to the image header

coi The channel of interest. 0 - all channels are selected, 1 - first channel is selected, etc. Note that the channel indices become 1-based.

If the ROI is set to NULL and the coi is *not* 0, the ROI is allocated. Most OpenCV functions do *not* support the COI setting, so to process an individual image/matrix channel one may copy (via cvCopy or cvSplit) the channel to a separate image/matrix, process it and then copy the result back (via cvCopy or cvMerge) if needed.

---

cvSetImageROI

Sets an image Region Of Interest (ROI) for a given rectangle.

```
void cvSetImageROI(
           IplImage* image,
           CvRect rect);
```

image A pointer to the image header

rect The ROI rectangle

If the original image ROI was `NULL` and the `rect` is not the whole image, the ROI structure is allocated.

Most OpenCV functions support the use of ROI and treat the image rectangle as a separate image. For example, all of the pixel coordinates are counted from the top-left (or bottom-left) corner of the ROI, not the original image.

---

cvSetReal1D

Set a specific array element.

arr Input array

idx Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 1.

---

cvSetReal2D

Set a specific array element.

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

value The value to assign to the element

Sets a specific array element. Array must have dimension 2.

---

cvSetReal3D

Set a specific array element.

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 3.

cvSetRealND

Set a specific array element.

arr Input array

indices List of zero-based element indices

value The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

cvSetZero

Clears the array.

```
void cvSetZero(CvArr* arr);
```

```
#define cvZero cvSetZero
```

arr Array to be cleared

The function clears the array. In the case of dense arrays (CvMat, CvMatND or IplImage), cvZero(array) is equivalent to cvSet(array,cvScalarAll(0),0). In the case of sparse arrays all the elements are removed.

cvSolve

Solves a linear system or least-squares problem.

```
int cvSolve(const CvArr* src1, const CvArr* src2, CvArr* dst, int
method=CV_LU);
```

A The source matrix

B The right-hand part of the linear system

X The output solution

method The solution (matrix inversion) method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix.

The function solves a linear system or least-squares problem (the latter is possible with SVD methods):

$$\texttt{dst} = argmin_X ||\texttt{src1}\,\texttt{X} - \texttt{src2}||$$

If CV_LU method is used, the function returns 1 if src1 is non-singular and 0 otherwise; in the latter case dst is not valid.

---

cvSolveCubic

Finds the real roots of a cubic equation.

```
void cvSolveCubic(const CvArr* coeffs, CvArr* roots);
```

coeffs The equation coefficients, an array of 3 or 4 elements

roots The output array of real roots which should have 3 elements

The function finds the real roots of a cubic equation:
If coeffs is a 4-element vector:

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or if coeffs is 3-element vector:

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

The function returns the number of real roots found. The roots are stored to root array, which is padded with zeros if there is only one root.

cvSplit

Divides multi-channel array into several single-channel arrays or extracts a single channel from the array.

```
void cvSplit(const CvArr* src, CvArr* dst0, CvArr* dst1, CvArr* dst2,
CvArr* dst3);
```

```
#define cvCvtPixToPlane cvSplit
```

src Source array

dst0 Destination channel 0

dst1 Destination channel 1

dst2 Destination channel 2

dst3 Destination channel 3

The function divides a multi-channel array into separate single-channel arrays. Two modes are available for the operation. If the source array has N channels then if the first N destination channels are not NULL, they all are extracted from the source array; if only a single destination channel of the first N is not NULL, this particular channel is extracted; otherwise an error is raised. The rest of the destination channels (beyond the first N) must always be NULL. For IplImage cvCopy with COI set can be also used to extract a single channel from the image.

cvSqrt

Calculates the square root.

```
float cvSqrt(float value);
```

value The input floating-point value

The function calculates the square root of the argument. If the argument is negative, the result is not determined.

cvSub

Computes the per-element difference between two arrays.

```
void cvSub(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts one array from another one:

```
dst(I)=src1(I)-src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cvSubRS

Computes the difference between a scalar and an array.

```
void cvSubRS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The first source array

value Scalar to subtract from

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts every element of source array from a scalar:

```
dst(I)=value-src(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvSubS

Computes the difference between an array and a scalar.

```
void cvSubS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Subtracted scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts a scalar from every element of the source array:

```
dst(I)=src(I)-value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cvSum

Adds up array elements.

```
CvScalar cvSum(const CvArr* arr);
```

arr The array

The function calculates the sum `S` of array elements, independently for each channel:

$$\sum_I \texttt{arr}(I)_c$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the sum to the first scalar component.

cvSVBkSb

Performs singular value back substitution.

```
void cvSVBkSb(
            const CvArr* W,
            const CvArr* U,
            const CvArr* V,
            const CvArr* B,
            CvArr* X,
            int flags);
```

W Matrix or vector of singular values

U Left orthogonal matrix (tranposed, perhaps)

V Right orthogonal matrix (tranposed, perhaps)

B The matrix to multiply the pseudo-inverse of the original matrix `A` by. This is an optional parameter. If it is omitted then it is assumed to be an identity matrix of an appropriate size (so that `X` will be the reconstructed pseudo-inverse of `A`).

X The destination matrix: result of back substitution

flags Operation flags, should match exactly to the `flags` passed to cvSVD

The function calculates back substitution for decomposed matrix `A` (see cvSVD description) and matrix `B`:

$$\mathtt{X} = \mathtt{V}\mathtt{W}^{-1}\mathtt{U}^T\mathtt{B}$$

where

$$W^{-1}_{(i,i)} = \begin{cases} 1/W_{(i,i)} & \text{if } W_{(i,i)} > \epsilon \sum_i W_{(i,i)} \\ 0 & \text{otherwise} \end{cases}$$

and ϵ is a small number that depends on the matrix data type.

This function together with cvSVD is used inside cvInvert and cvSolve, and the possible reason to use these (svd and bksb) "low-level" function, is to avoid allocation of temporary matrices inside the high-level counterparts (inv and solve).

cvSVD

Performs singular value decomposition of a real floating-point matrix.

```
void cvSVD(
           CvArr* A,
           CvArr* W,
           CvArr* U=NULL,
           CvArr* V=NULL,
           int flags=0);
```

A Source $\mathtt{M} \times \mathtt{N}$ matrix

W Resulting singular value diagonal matrix ($\mathtt{M} \times \mathtt{N}$ or $\min(\mathtt{M},\mathtt{N}) \times \min(\mathtt{M},\mathtt{N})$) or $\min(\mathtt{M},\mathtt{N}) \times 1$ vector of the singular values

U Optional left orthogonal matrix, $\mathtt{M} \times \min(\mathtt{M},\mathtt{N})$ (when CV_SVD_U_T is not set), or $\min(\mathtt{M},\mathtt{N}) \times \mathtt{M}$ (when CV_SVD_U_T is set), or $\mathtt{M} \times \mathtt{M}$ (regardless of CV_SVD_U_T flag).

V Optional right orthogonal matrix, $\mathtt{N} \times \min(\mathtt{M},\mathtt{N})$ (when CV_SVD_V_T is not set), or $\min(\mathtt{M},\mathtt{N}) \times \mathtt{N}$ (when CV_SVD_V_T is set), or $\mathtt{N} \times \mathtt{N}$ (regardless of CV_SVD_V_T flag).

flags Operation flags; can be 0 or a combination of the following values:

- **CV_SVD_MODIFY_A** enables modification of matrix A during the operation. It speeds up the processing.
- **CV_SVD_U_T** means that the transposed matrix U is returned. Specifying the flag speeds up the processing.
- **CV_SVD_V_T** means that the transposed matrix V is returned. Specifying the flag speeds up the processing.

The function decomposes matrix A into the product of a diagonal matrix and two orthogonal matrices:

$$A = U\,W\,V^T$$

where W is a diagonal matrix of singular values that can be coded as a 1D vector of singular values and U and V. All the singular values are non-negative and sorted (together with U and V columns) in descending order.

An SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix A is a square, symmetric, and positively defined matrix, for example, when it is a covariance matrix. W in this case will be a vector/-matrix of the eigenvalues, and $U = V$ will be a matrix of the eigenvectors.
- accurate solution of a poor-conditioned linear system.
- least-squares solution of an overdetermined linear system. This and the preceeding is done by using the cvSolve function with the CV_SVD method.
- accurate calculation of different matrix characteristics such as the matrix rank (the number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), and determinant (absolute value of the determinant is equal to the product of singular values).

---

cvTrace

Returns the trace of a matrix.

```
CvScalar cvTrace(const CvArr* mat);
```

mat The source matrix

The function returns the sum of the diagonal elements of the matrix src1.

$$tr(\texttt{mat}) = \sum_i \texttt{mat}(i,i)$$

---

cvTransform

Performs matrix transformation of every array element.

```
void cvTransform(const CvArr* src, CvArr* dst, const CvMat* transmat,
const CvMat* shiftvec=NULL);
```

src The first source array

dst The destination array

transmat Transformation matrix

shiftvec Optional shift vector

The function performs matrix transformation of every element of array `src` and stores the results in `dst`:

$$dst(I) = transmat \cdot src(I) + shiftvec$$

That is, every element of an `N`-channel array `src` is considered as an `N`-element vector which is transformed using a `M` × `N` matrix `transmat` and shift vector `shiftvec` into an element of `M`-channel array `dst`. There is an option to embedd `shiftvec` into `transmat`. In this case `transmat` should be a `M` × $(N + 1)$ matrix and the rightmost column is treated as the shift vector.

Both source and destination arrays should have the same depth and the same size or selected ROI size. `transmat` and `shiftvec` should be real floating-point matrices.

The function may be used for geometrical transformation of n dimensional point set, arbitrary linear color space transformation, shuffling the channels and so forth.

---

cvTranspose

Transposes a matrix.

```
void cvTranspose(const CvArr* src, CvArr* dst);
```

```
#define cvT cvTranspose
```

src The source matrix

dst The destination matrix

The function transposes matrix `src1`:

$$\mathtt{dst}(i,j) = \mathtt{src}(j,i)$$

Note that no complex conjugation is done in the case of a complex matrix. Conjugation should be done separately: look at the sample code in cvXorS for an example.

cvXor

Performs per-element bit-wise ”exclusive or” operation on two arrays.

```
void cvXor(const CvArr* src1, const CvArr* src2, CvArr* dst, const
CvArr* mask=NULL);
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)^src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

cvXorS

Performs per-element bit-wise ”exclusive or” operation on an array and a scalar.

```
void cvXorS(const CvArr* src, CvScalar value, CvArr* dst, const CvArr*
mask=NULL);
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function XorS calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)^value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

The following sample demonstrates how to conjugate complex vector by switching the most-significant bit of imaging part:

```
float a[] = { 1, 0, 0, 1, -1, 0, 0, -1 }; /* 1, j, -1, -j */
CvMat A = cvMat(4, 1, CV\_32FC2, &a);
int i, negMask = 0x80000000;
cvXorS(&A, cvScalar(0, *(float*)&negMask, 0, 0 ), &A, 0);
for(i = 0; i < 4; i++ )
    printf("(%.1f, %.1f) ", a[i*2], a[i*2+1]);
```

The code should print:

```
(1.0,0.0) (0.0,-1.0) (-1.0,0.0) (0.0,1.0)
```

---

cvmGet

Returns the particular element of single-channel floating-point matrix.

```
double cvmGet(const CvMat* mat, int row, int col);
```

mat Input matrix

row The zero-based index of row

col The zero-based index of column

The function is a fast replacement for cvGetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

---

cvmSet

Returns a specific element of a single-channel floating-point matrix.

```
void cvmSet(CvMat* mat, int row, int col, double value);
```

mat The matrix

row The zero-based index of row

col The zero-based index of column

value The new value of the matrix element

The function is a fast replacement for cvSetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

1.3 Dynamic Structures

CvMemStorage

Growing memory storage.

```
typedef struct CvMemStorage
{
    struct CvMemBlock* bottom;/* first allocated block */
    struct CvMemBlock* top; /* the current memory block - top of the stack */
    struct CvMemStorage* parent; /* borrows new blocks from */
    int block\_size; /* block size */
    int free\_space; /* free space in the \texttt{top} block (in bytes) */
} CvMemStorage;
```

Memory storage is a low-level structure used to store dynamicly growing data structures such as sequences, contours, graphs, subdivisions, etc. It is organized as a list of memory blocks of equal size - `bottom` field is the beginning of the list of blocks and `top` is the currently used block, but not necessarily the last block of the list. All blocks between `bottom` and `top`, not including the latter, are considered fully occupied; all blocks between `top` and the last block, not including `top`, are considered free and `top` itself is partly ocupied - `free_space` contains the number of free bytes left in the end of `top`.

A new memory buffer that may be allocated explicitly by cvMemStorageAlloc function or implicitly by higher-level functions, such as cvSeqPush, cvGraphAddEdge, etc., `always` starts in the end of the current block if it fits there. After allocation, `free_space` is decremented by the size of the allocated buffer plus some padding to keep the proper alignment. When the allocated

buffer does not fit into the available portion of `top`, the next storage block from the list is taken as `top` and `free_space` is reset to the whole block size prior to the allocation.

If there are no more free blocks, a new block is allocated (or borrowed from the parent, see cvCreateChildMemStorage) and added to the end of list. Thus, the storage behaves as a stack with `bottom` indicating bottom of the stack and the pair (`top`, `free_space`) indicating top of the stack. The stack top may be saved via cvSaveMemStoragePos, restored via cvRestoreMemStoragePos, or reset via cvClearStorage.

CvMemBlock

Memory storage block.

```
typedef struct CvMemBlock
{
    struct CvMemBlock* prev;
    struct CvMemBlock* next;
} CvMemBlock;
```

The structure CvMemBlock represents a single block of memory storage. The actual data in the memory blocks follows the header, that is, the i_{th} byte of the memory block can be retrieved with the expression `((char*)(mem_block_ptr+1))[i]`. However, there is normally no need to access the storage structure fields directly.

CvMemStoragePos

Memory storage position.

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
    int free\_space;
} CvMemStoragePos;
```

The structure described above stores the position of the stack top that can be saved via cvSaveMemStoragePos and restored via cvRestoreMemStoragePos.

CvSeq

Growable sequence of elements.

```
#define CV_SEQUENCE\_FIELDS() \
    int flags; /* micsellaneous flags */ \
    int header_size; /* size of sequence header */ \
```

```
    struct CvSeq* h_prev; /* previous sequence */ \
    struct CvSeq* h_next; /* next sequence */ \
    struct CvSeq* v_prev; /* 2nd previous sequence */ \
    struct CvSeq* v_next; /* 2nd next sequence */ \
    int total; /* total number of elements */ \
    int elem_size;/* size of sequence element in bytes */ \
    char* block_max;/* maximal bound of the last block */ \
    char* ptr; /* current write pointer */ \
    int delta_elems; /* how many elements allocated when the sequence grows
                        (sequence granularity) */ \
    CvMemStorage* storage; /* where the seq is stored */ \
    CvSeqBlock* free_blocks; /* free blocks list */ \
    CvSeqBlock* first; /* pointer to the first sequence block */

typedef struct CvSeq
{
    CV_SEQUENCE_FIELDS()
} CvSeq;
```

The structure CvSeq is a base for all of OpenCV dynamic data structures.

Such an unusual definition via a helper macro simplifies the extension of the structure CvSeq with additional parameters. To extend CvSeq the user may define a new structure and put user-defined fields after all CvSeq fields that are included via the macro `CV_SEQUENCE_FIELDS()`.

There are two types of sequences - dense and sparse. The base type for dense sequences is CvSeq and such sequences are used to represent growable 1d arrays - vectors, stacks, queues, and deques. They have no gaps in the middle - if an element is removed from the middle or inserted into the middle of the sequence, the elements from the closer end are shifted. Sparse sequences have CvSet as a base class and they are discussed later in more detail. They are sequences of nodes; each may be either occupied or free as indicated by the node flag. Such sequences are used for unordered data structures such as sets of elements, graphs, hash tables and so forth.

The field `header_size` contains the actual size of the sequence header and should be greater than or equal to `sizeof(CvSeq)`.

The fields `h_prev`, `h_next`, `v_prev`, `v_next` can be used to create hierarchical structures from separate sequences. The fields `h_prev` and `h_next` point to the previous and the next sequences on the same hierarchical level, while the fields `v_prev` and `v_next` point to the previous and the next sequences in the vertical direction, that is, the parent and its first child. But these are just names and the pointers can be used in a different way.

The field `first` points to the first sequence block, whose structure is described below.

The field `total` contains the actual number of dense sequence elements and number of allocated nodes in a sparse sequence.

The field `flags` contains the particular dynamic type signature (`CV_SEQ_MAGIC_VAL` for dense sequences and `CV_SET_MAGIC_VAL` for sparse sequences) in the highest 16 bits and miscellaneous information about the sequence. The lowest `CV_SEQ_ELTYPE_BITS` bits contain the ID of the element type. Most of sequence processing functions do not use element type but rather element size stored in `elem_size`. If a sequence contains the numeric data for one of the CvMat type then the element type matches to the corresponding CvMat element type, e.g., `CV_32SC2` may be used for a sequence of 2D points, `CV_32FC1` for sequences of floating-point values, etc. A `CV_SEQ_ELTYPE(seq_header_ptr)` macro retrieves the type of sequence elements. Processing functions that work with numerical sequences check that `elem_size` is equal to that calculated from the type element size. Besides CvMat compatible types, there are few extra element types defined in the `cvtypes.h` header:

Standard Types of Sequence Elements

```
#define CV_SEQ_ELTYPE_POINT          CV_32SC2  /* (x,y) */
#define CV_SEQ_ELTYPE_CODE           CV_8UC1   /* freeman code: 0..7 */
#define CV_SEQ_ELTYPE_GENERIC        0 /* unspecified type of
                                        sequence elements */
#define CV_SEQ_ELTYPE_PTR            CV_USRTYPE1 /* =6 */
#define CV_SEQ_ELTYPE_PPOINT         CV_SEQ_ELTYPE_PTR  /* &elem: pointer to
                                                  element of other sequence */
#define CV_SEQ_ELTYPE_INDEX          CV_32SC1  /* #elem: index of element of
                                                      some other sequence */
#define CV_SEQ_ELTYPE_GRAPH_EDGE     CV_SEQ_ELTYPE_GENERIC  /* &next_o,
                                                 &next_d, &vtx_o, &vtx_d */
#define CV_SEQ_ELTYPE_GRAPH_VERTEX   CV_SEQ_ELTYPE_GENERIC  /* first_edge,
                                                        &(x,y) */
#define CV_SEQ_ELTYPE_TRIAN_ATR      CV_SEQ_ELTYPE_GENERIC  /* vertex of the
                                                     binary tree   */
#define CV_SEQ_ELTYPE_CONNECTED_COMP CV_SEQ_ELTYPE_GENERIC  /* connected
                                                        component  */
#define CV_SEQ_ELTYPE_POINT3D        CV_32FC3  /* (x,y,z)  */
```

The next `CV_SEQ_KIND_BITS` bits specify the kind of sequence:

Standard Kinds of Sequences

```
/* generic (unspecified) kind of sequence */
#define CV_SEQ_KIND_GENERIC     (0 << CV_SEQ_ELTYPE_BITS)

/* dense sequence suntypes */
#define CV_SEQ_KIND_CURVE       (1 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_BIN_TREE    (2 << CV_SEQ_ELTYPE_BITS)

/* sparse sequence (or set) subtypes */
```

```
#define CV_SEQ_KIND_GRAPH       (3 << CV_SEQ_ELTYPE_BITS)
#define CV_SEQ_KIND_SUBDIV2D    (4 << CV_SEQ_ELTYPE_BITS)
```

The remaining bits are used to identify different features specific to certain sequence kinds and element types. For example, curves made of points (CV_SEQ_KIND_CURVE|CV_SEQ_ELTYPE_POINT), together with the flag CV_SEQ_FLAG_CLOSED, belong to the type CV_SEQ_POLYGON or, if other flags are used, to its subtype. Many contour processing functions check the type of the input sequence and report an error if they do not support this type. The file cvtypes.h stores the complete list of all supported predefined sequence types and helper macros designed to get the sequence type of other properties. The definition of the building blocks of sequences can be found below.

CvSeqBlock

Continuous sequence block.

```
typedef struct CvSeqBlock
{
    struct CvSeqBlock* prev; /* previous sequence block */
    struct CvSeqBlock* next; /* next sequence block */
    int start_index; /* index of the first element in the block +
    sequence->first->start_index */
    int count; /* number of elements in the block */
    char* data; /* pointer to the first element of the block */
} CvSeqBlock;
```

Sequence blocks make up a circular double-linked list, so the pointers prev and next are never NULL and point to the previous and the next sequence blocks within the sequence. It means that next of the last block is the first block and prev of the first block is the last block. The fields startIndex and count help to track the block location within the sequence. For example, if the sequence consists of 10 elements and splits into three blocks of 3, 5, and 2 elements, and the first block has the parameter startIndex = 2, then pairs (startIndex, count) for the sequence blocks are (2,3), (5, 5), and (10, 2) correspondingly. The parameter startIndex of the first block is usually 0 unless some elements have been inserted at the beginning of the sequence.

CvSlice

A sequence slice.

```
typedef struct CvSlice
{
    int start_index;
    int end_index;
} CvSlice;
```

```
inline CvSlice cvSlice( int start, int end );
#define CV_WHOLE_SEQ_END_INDEX 0x3fffffff
#define CV_WHOLE_SEQ  cvSlice(0, CV_WHOLE_SEQ_END_INDEX)

/* calculates the sequence slice length */
int cvSliceLength( CvSlice slice, const CvSeq* seq );
```

Some of functions that operate on sequences take a `CvSlice slice` parameter that is often set to the whole sequence (CV_WHOLE_SEQ) by default. Either of the `startIndex` and `endIndex` may be negative or exceed the sequence length, `startIndex` is inclusive, and `endIndex` is an exclusive boundary. If they are equal, the slice is considered empty (i.e., contains no elements). Because sequences are treated as circular structures, the slice may select a few elements in the end of a sequence followed by a few elements at the beginning of the sequence. For example, `cvSlice(-2, 3)` in the case of a 10-element sequence will select a 5-element slice, containing the pre-last (8th), last (9th), the very first (0th), second (1th) and third (2nd) elements. The functions normalize the slice argument in the following way: first, cvSliceLength is called to determine the length of the slice, then, `startIndex` of the slice is normalized similarly to the argument of cvGetSeqElem (i.e., negative indices are allowed). The actual slice to process starts at the normalized `startIndex` and lasts cvSliceLength elements (again, assuming the sequence is a circular structure).

If a function does not accept a slice argument, but you want to process only a part of the sequence, the sub-sequence may be extracted using the cvSeqSlice function, or stored into a continuous buffer with CvtSeqToArray (optionally, followed by cvMakeSeqHeaderForArray).

CvSet

Collection of nodes.

```
typedef struct CvSetElem
{
    int flags; /* it is negative if the node is free and zero or positive otherwise */
    struct CvSetElem* next_free; /* if the node is free, the field is a
                                    pointer to next free node */
}
CvSetElem;

#define CV_SET_FIELDS()    \
    CV_SEQUENCE_FIELDS()   /* inherits from [#CvSeq CvSeq] */ \
    struct CvSetElem* free_elems; /* list of free nodes */

typedef struct CvSet
{
```

```
    CV_SET_FIELDS()
} CvSet;
```

The structure CvSet is a base for OpenCV sparse data structures.

As follows from the above declaration, CvSet inherits from CvSeq and it adds the `free_elems` field, which is a list of free nodes, to it. Every set node, whether free or not, is an element of the underlying sequence. While there are no restrictions on elements of dense sequences, the set (and derived structures) elements must start with an integer field and be able to fit CvSetElem structure, because these two fields (an integer followed by a pointer) are required for the organization of a node set with the list of free nodes. If a node is free, the `flags` field is negative (the most-significant bit, or MSB, of the field is set), and the `next_free` points to the next free node (the first free node is referenced by the `free_elems` field of CvSet). And if a node is occupied, the `flags` field is positive and contains the node index that may be retrieved using the (`set_elem->flags & CV_SET_ELEM_IDX_MASK`) expressions, the rest of the node content is determined by the user. In particular, the occupied nodes are not linked as the free nodes are, so the second field can be used for such a link as well as for some different purpose. The macro `CV_IS_SET_ELEM(set_elem_ptr)` can be used to determined whether the specified node is occupied or not.

Initially the set and the list are empty. When a new node is requested from the set, it is taken from the list of free nodes, which is then updated. If the list appears to be empty, a new sequence block is allocated and all the nodes within the block are joined in the list of free nodes. Thus, the `total` field of the set is the total number of nodes both occupied and free. When an occupied node is released, it is added to the list of free nodes. The node released last will be occupied first.

In OpenCV CvSet is used for representing graphs (CvGraph), sparse multi-dimensional arrays (CvSparseMat), and planar subdivisions CvSubdiv2D .

CvGraph

Oriented or unoriented weighted graph.

```
#define CV_GRAPH_VERTEX_FIELDS()    \
    int flags; /* vertex flags */   \
    struct CvGraphEdge* first; /* the first incident edge */

typedef struct CvGraphVtx
{
    CV_GRAPH_VERTEX_FIELDS()
}
CvGraphVtx;

#define CV_GRAPH_EDGE_FIELDS()      \
    int flags; /* edge flags */     \
```

```
    float weight; /* edge weight */ \
    struct CvGraphEdge* next[2]; /* the next edges in the incidence lists for staring (0) a
                                    /* and ending (1) vertices */ \
    struct CvGraphVtx* vtx[2]; /* the starting (0) and ending (1) vertices */

typedef struct CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS()
}
CvGraphEdge;

#define  CV_GRAPH_FIELDS()                  \
    CV_SET_FIELDS() /* set of vertices */   \
    CvSet* edges;   /* set of edges */

typedef struct CvGraph
{
    CV_GRAPH_FIELDS()
}
CvGraph;
```

The structure CvGraph is a base for graphs used in OpenCV.

The graph structure inherits from CvSet - which describes common graph properties and the graph vertices, and contains another set as a member - which describes the graph edges.

The vertex, edge, and the graph header structures are declared using the same technique as other extendible OpenCV structures - via macros, which simplify extension and customization of the structures. While the vertex and edge structures do not inherit from CvSetElem explicitly, they satisfy both conditions of the set elements: having an integer field in the beginning and fitting within the CvSetElem structure. The `flags` fields are used as for indicating occupied vertices and edges as well as for other purposes, for example, for graph traversal (see cvCreateGraphScanner et al.), so it is better not to use them directly.

The graph is represented as a set of edges each of which has a list of incident edges. The incidence lists for different vertices are interleaved to avoid information duplication as much as posssible.

The graph may be oriented or unoriented. In the latter case there is no distiction between the edge connecting vertex A with vertex B and the edge connecting vertex B with vertex A - only one of them can exist in the graph at the same moment and it represents both $A \rightarrow B$ and $B \rightarrow A$ edges.

CvGraphScanner

Graph traversal state.

```
typedef struct CvGraphScanner
{
    CvGraphVtx* vtx;       /* current graph vertex (or current edge origin) */
    CvGraphVtx* dst;       /* current graph edge destination vertex */
    CvGraphEdge* edge;     /* current edge */

    CvGraph* graph;        /* the graph */
    CvSeq*   stack;        /* the graph vertex stack */
    int      index;        /* the lower bound of certainly visited vertices */
    int      mask;         /* event mask */
}
CvGraphScanner;
```

The structure CvGraphScanner is used for depth-first graph traversal. See discussion of the functions below.

CV_TREE_NODE_FIELDS

Helper macro for a tree node type declaration.

The macro `CV_TREE_NODE_FIELDS()` is used to declare structures that can be organized into hierarchical strucutures (trees), such as CvSeq - the basic type for all dynamic structures. The trees created with nodes declared using this macro can be processed using the functions described below in this section.

CvTreeNodeIterator

Opens existing or creates new file storage.

```
typedef struct CvTreeNodeIterator
{
    const void* node;
    int level;
    int max_level;
}
CvTreeNodeIterator;
```

```
#define CV_TREE_NODE_FIELDS(node_type)                          \
    int       flags;         /* micsellaneous flags */          \
    int       header_size;   /* size of sequence header */      \
    struct    node_type* h_prev; /* previous sequence */        \
    struct    node_type* h_next; /* next sequence */            \
    struct    node_type* v_prev; /* 2nd previous sequence */    \
    struct    node_type* v_next; /* 2nd next sequence */
```

The structure CvTreeNodeIterator is used to traverse trees. Each tree node should start with the certain fields which are defined by `CV_TREE_NODE_FIELDS(...)` macro. In C++ terms, each tree node should be a structure "derived" from

```
struct _BaseTreeNode
{
    CV_TREE_NODE_FIELDS(_BaseTreeNode);
}
```

`CvSeq`, `CvSet`, `CvGraph` and other dynamic structures derived from `CvSeq` comply with the requirement.

---

cvClearGraph

Clears a graph.

```
void cvClearGraph( CvGraph* graph );
```

graph Graph

The function removes all vertices and edges from a graph. The function has O(1) time complexity.

---

cvClearMemStorage

Clears memory storage.

```
void cvClearMemStorage( CvMemStorage* storage );
```

storage Memory storage

The function resets the top (free space boundary) of the storage to the very beginning. This function does not deallocate any memory. If the storage has a parent, the function returns all blocks to the parent.

cvClearSeq

Clears a sequence.

```
void cvClearSeq( CvSeq* seq );
```

seq Sequence

The function removes all elements from a sequence. The function does not return the memory to the storage block, but this memory is reused later when new elements are added to the sequence. The function has 'O(1)' time complexity.

cvClearSet

Clears a set.

```
void cvClearSet( CvSet* setHeader );
```

setHeader Cleared set

The function removes all elements from set. It has O(1) time complexity.

cvCloneGraph

Clones a graph.

```
CvGraph* cvCloneGraph(
           const CvGraph* graph,
           CvMemStorage* storage );
```

graph The graph to copy

storage Container for the copy

The function creates a full copy of the specified graph. If the graph vertices or edges have pointers to some external data, it can still be shared between the copies. The vertex and edge indices in the new graph may be different from the original because the function defragments the vertex and edge sets.

cvCloneSeq

Creates a copy of a sequence.

```
CvSeq* cvCloneSeq(
           const CvSeq* seq,
           CvMemStorage* storage=NULL );
```

seq Sequence

storage The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

The function makes a complete copy of the input sequence and returns it.
The call

```
cvCloneSeq( seq, storage )
```

is equivalent to

```
cvSeqSlice( seq, CV_WHOLE_SEQ, storage, 1 )
```

cvCreateChildMemStorage

Creates child memory storage.

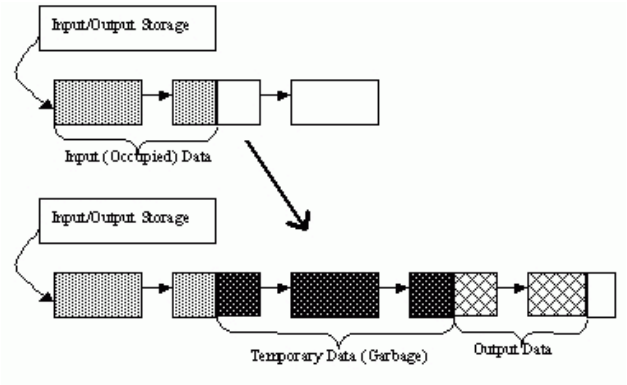
```
CvMemStorage* cvCreateChildMemStorage(CvMemStorage* parent);
```

parent Parent memory storage

The function creates a child memory storage that is similar to simple memory storage except for the differences in the memory allocation/deallocation mechanism. When a child storage needs a new block to add to the block list, it tries to get this block from the parent. The first unoccupied parent block available is taken and excluded from the parent block list. If no blocks are available, the parent either allocates a block or borrows one from its own parent, if any. In other words, the chain, or a more complex structure, of memory storages where every storage is a child/parent of another is possible. When a child storage is released or even cleared, it returns all blocks to the parent. In other aspects, child storage is the same as simple storage.

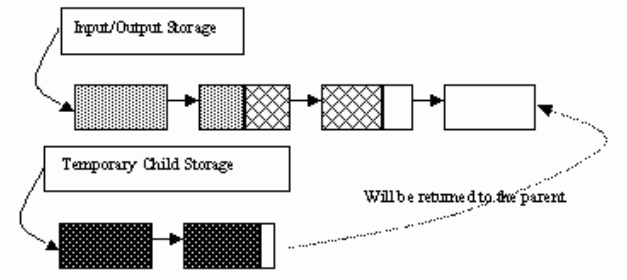
Child storage is useful in the following situation. Imagine that the user needs to process dynamic data residing in a given storage area and put the result back to that same storage area. With the simplest approach, when temporary data is resided in the same storage area as the input and output data, the storage area will look as follows after processing:

Dynamic data processing without using child storage



That is, garbage appears in the middle of the storage. However, if one creates a child memory storage at the beginning of processing, writes temporary data there, and releases the child storage at the end, no garbage will appear in the source/destination storage:

Dynamic data processing using a child storage



---

cvCreateGraph

Creates an empty graph.

```
CvGraph* cvCreateGraph(
            int graph_flags,
            int header_size,
            int vtx_size,
            int edge_size,
            CvMemStorage* storage );
```

graph_flags Type of the created graph. Usually, it is either `CV_SEQ_KIND_GRAPH` for generic unoriented graphs and `CV_SEQ_KIND_GRAPH | CV_GRAPH_FLAG_ORIENTED` for generic oriented graphs.

header_size Graph header size; may not be less than `sizeof(CvGraph)`

vtx_size Graph vertex size; the custom vertex structure must start with CvGraphVtx (use `CV_GRAPH_VERTEX_FIELDS()`)

edge_size Graph edge size; the custom edge structure must start with CvGraphEdge (use `CV_GRAPH_EDGE_FIELDS()`)

storage The graph container

The function creates an empty graph and returns a pointer to it.

---

cvCreateGraphScanner

Creates structure for depth-first graph traversal.

```
CvGraphScanner* cvCreateGraphScanner(
            CvGraph* graph,
            CvGraphVtx* vtx=NULL,
            int mask=CV_GRAPH_ALL_ITEMS );
```

graph Graph

vtx Initial vertex to start from. If NULL, the traversal starts from the first vertex (a vertex with the minimal index in the sequence of vertices).

mask Event mask indicating which events are of interest to the user (where cvNextGraphItem function returns control to the user) It can be `CV_GRAPH_ALL_ITEMS` (all events are of interest) or a combination of the following flags:

- **CV_GRAPH_VERTEX** stop at the graph vertices visited for the first time
- **CV_GRAPH_TREE_EDGE** stop at tree edges (`tree edge` is the edge connecting the last visited vertex and the vertex to be visited next)
- **CV_GRAPH_BACK_EDGE** stop at back edges (`back edge` is an edge connecting the last visited vertex with some of its ancestors in the search tree)

CV_GRAPH_FORWARD_EDGE stop at forward edges (`forward edge` is an edge conecting the last visited vertex with some of its descendants in the search tree. The forward edges are only possible during oriented graph traversal)

CV_GRAPH_CROSS_EDGE stop at cross edges (`cross edge` is an edge connecting different search trees or branches of the same tree. The `cross edges` are only possible during oriented graph traversal)

CV_GRAPH_ANY_EDGE stop at any edge (`tree, back, forward`, and `cross edges`)

CV_GRAPH_NEW_TREE stop in the beginning of every new search tree. When the traversal procedure visits all vertices and edges reachable from the initial vertex (the visited vertices together with tree edges make up a tree), it searches for some unvisited vertex in the graph and resumes the traversal process from that vertex. Before starting a new tree (including the very first tree when `cvNextGraphItem` is called for the first time) it generates a `CV_GRAPH_NEW_TREE` event. For unoriented graphs, each search tree corresponds to a connected component of the graph.

CV_GRAPH_BACKTRACKING stop at every already visited vertex during backtracking - returning to already visited vertexes of the traversal tree.

The function creates a structure for depth-first graph traversal/search. The initialized structure is used in the cvNextGraphItem function - the incremental traversal procedure.

---

cvCreateMemStorage

Creates memory storage.

```
CvMemStorage* cvCreateMemStorage( int blockSize=0 );
```

blockSize Size of the storage blocks in bytes. If it is 0, the block size is set to a default value - currently it is about 64K.

The function creates an empty memory storage. See CvMemStorage description.

---

cvCreateSeq

Creates a sequence.

```
CvSeq* cvCreateSeq(
           int seqFlags,
           int headerSize,
           int elemSize,
           CvMemStorage* storage);
```

seqFlags Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be set to 0, otherwise the appropriate type must be selected from the list of predefined sequence types.

headerSize Size of the sequence header; must be greater than or equal to `sizeof(CvSeq)`. If a specific type or its extension is indicated, this type must fit the base type header.

elemSize Size of the sequence elements in bytes. The size must be consistent with the sequence type. For example, for a sequence of points to be created, the element type `CV_SEQ_ELTYPE_POINT` should be specified and the parameter `elemSize` must be equal to `sizeof(CvPoint)`.

storage Sequence location

The function creates a sequence and returns the pointer to it. The function allocates the sequence header in the storage block as one continuous chunk and sets the structure fields `flags`, `elemSize`, `headerSize`, and `storage` to passed values, sets `delta_elems` to the default value (that may be reassigned using the cvSetSeqBlockSize function), and clears other header fields, including the space following the first `sizeof(CvSeq)` bytes.

---

cvCreateSet

Creates an empty set.

```
CvSet* cvCreateSet(
           int set_flags,
           int header_size,
           int elem_size,
           CvMemStorage* storage );
```

set_flags Type of the created set

header_size Set header size; may not be less than `sizeof(CvSet)`

elem_size Set element size; may not be less than CvSetElem

storage Container for the set

The function creates an empty set with a specified header size and element size, and returns the pointer to the set. This function is just a thin layer on top of cvCreateSeq.

---

cvCvtSeqToArray

Copies a sequence to one continuous block of memory.

```
void* cvCvtSeqToArray(
           const CvSeq* seq,
           void* elements,
           CvSlice slice=CV_WHOLE_SEQ );
```

seq Sequence

elements Pointer to the destination array that must be large enough. It should be a pointer to data, not a matrix header.

slice The sequence portion to copy to the array

The function copies the entire sequence or subsequence to the specified buffer and returns the pointer to the buffer.

---

cvEndWriteSeq

Finishes the process of writing a sequence.

```
CvSeq* cvEndWriteSeq( CvSeqWriter* writer );
```

writer Writer state

The function finishes the writing process and returns the pointer to the written sequence. The function also truncates the last incomplete sequence block to return the remaining part of the block to memory storage. After that, the sequence can be read and modified safely. See cvcvStartWriteSeq and cvcvStartAppendToSeq

cvFindGraphEdge

Finds an edge in a graph.

```
CvGraphEdge* cvFindGraphEdge( const CvGraph* graph, int start_idx, int
end_idx );
```

```
#define cvGraphFindEdge cvFindGraphEdge
```

graph Graph

start_idx Index of the starting vertex of the edge

end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function finds the graph edge connecting two specified vertices and returns a pointer to it or NULL if the edge does not exist.

cvFindGraphEdgeByPtr

Finds an edge in a graph by using its pointer.

```
CvGraphEdge* cvFindGraphEdgeByPtr(
           const CvGraph* graph,
           const CvGraphVtx* startVtx,
           const CvGraphVtx* endVtx );
```

```
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

graph Graph

startVtx Pointer to the starting vertex of the edge

endVtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function finds the graph edge connecting two specified vertices and returns pointer to it or NULL if the edge does not exists.

cvFlushSeqWriter

Updates sequence headers from the writer.

```
void cvFlushSeqWriter( CvSeqWriter* writer );
```

writer Writer state

The function is intended to enable the user to read sequence elements, whenever required, during the writing process, e.g., in order to check specific conditions. The function updates the sequence headers to make reading from the sequence possible. The writer is not closed, however, so that the writing process can be continued at any time. If an algorithm requires frequent flushes, consider using cvSeqPush instead.

cvGetGraphVtx

Finds a graph vertex by using its index.

```
CvGraphVtx* cvGetGraphVtx(
            CvGraph* graph,
            int vtx_idx );
```

graph Graph

vtx_idx Index of the vertex

The function finds the graph vertex by using its index and returns the pointer to it or NULL if the vertex does not belong to the graph.

cvGetSeqElem

Returns a pointer to a sequence element according to its index.

```
char* cvGetSeqElem( const CvSeq* seq, int index );
```

```
#define CV_GET_SEQ_ELEM( TYPE, seq, index )  (TYPE*)cvGetSeqElem( (CvSeq*)(seq), (index) )
```

seq Sequence

index Index of element

The function finds the element with the given index in the sequence and returns the pointer to it. If the element is not found, the function returns 0. The function supports negative indices, where -1 stands for the last sequence element, -2 stands for the one before last, etc. If the sequence is most likely to consist of a single sequence block or the desired element is likely to be located in the first block, then the macro `CV_GET_SEQ_ELEM( elemType, seq, index )` should be used, where the parameter `elemType` is the type of sequence elements (CvPoint for example), the parameter `seq` is a sequence, and the parameter `index` is the index of the desired element. The macro checks first whether the desired element belongs to the first block of the sequence and returns it if it does; otherwise the macro calls the main function `GetSeqElem`. Negative indices always cause the cvGetSeqElem call. The function has O(1) time complexity assuming that the number of blocks is much smaller than the number of elements.

---

cvGetSeqReaderPos

Returns the current reader position.

```
int cvGetSeqReaderPos( CvSeqReader* reader );
```

reader Reader state

The function returns the current reader position (within 0 ... `reader->seq->total` - 1).

---

cvGetSetElem

Finds a set element by its index.

```
CvSetElem* cvGetSetElem(
           const CvSet* setHeader,
           int index );
```

setHeader Set

index Index of the set element within a sequence

The function finds a set element by its index. The function returns the pointer to it or 0 if the index is invalid or the corresponding node is free. The function supports negative indices as it uses cvGetSeqElem to locate the node.

---

cvGraphAddEdge

Adds an edge to a graph.

```
int cvGraphAddEdge(
            CvGraph* graph,
            int start_idx,
            int end_idx,
            const CvGraphEdge* edge=NULL,
            CvGraphEdge** inserted_edge=NULL );
```

graph Graph

start_idx Index of the starting vertex of the edge

end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

edge Optional input parameter, initialization data for the edge

inserted_edge Optional output parameter to contain the address of the inserted edge

The function connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already and -1 if either of the vertices was not found, the starting and the ending vertex are the same, or there is some other critical situation. In the latter case (i.e., when the result is negative), the function also reports an error by default.

---

cvGraphAddEdgeByPtr

Adds an edge to a graph by using its pointer.

```
int cvGraphAddEdgeByPtr(
           CvGraph* graph,
           CvGraphVtx* start_vtx,
           CvGraphVtx* end_vtx,
           const CvGraphEdge* edge=NULL,
           CvGraphEdge** inserted_edge=NULL );
```

graph Graph

start_vtx Pointer to the starting vertex of the edge

end_vtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

edge Optional input parameter, initialization data for the edge

inserted_edge Optional output parameter to contain the address of the inserted edge within the edge set

The function connects two specified vertices. The function returns 1 if the edge has been added successfully, 0 if the edge connecting the two vertices exists already, and -1 if either of the vertices was not found, the starting and the ending vertex are the same or there is some other critical situation. In the latter case (i.e., when the result is negative), the function also reports an error by default.

---

cvGraphAddVtx

Adds a vertex to a graph.

```
int cvGraphAddVtx(
           CvGraph* graph,
           const CvGraphVtx* vtx=NULL,
           CvGraphVtx** inserted_vtx=NULL );
```

graph Graph

vtx Optional input argument used to initialize the added vertex (only user-defined fields beyond `sizeof(CvGraphVtx)` are copied)

inserted_vertex Optional output argument. If not NULL, the address of the new vertex is written here.

The function adds a vertex to the graph and returns the vertex index.

---

cvGraphEdgeIdx

Returns the index of a graph edge.

```
int cvGraphEdgeIdx(
           CvGraph* graph,
           CvGraphEdge* edge );
```

graph Graph

edge Pointer to the graph edge

The function returns the index of a graph edge.

---

cvGraphRemoveEdge

Removes an edge from a graph.

```
void cvGraphRemoveEdge(
           CvGraph* graph,
           int start_idx,
           int end_idx );
```

graph Graph

start_idx Index of the starting vertex of the edge

end_idx Index of the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

cvGraphRemoveEdgeByPtr

Removes an edge from a graph by using its pointer.

```
void cvGraphRemoveEdgeByPtr(
           CvGraph* graph,
           CvGraphVtx* start_vtx,
           CvGraphVtx* end_vtx );
```

graph Graph

start_vtx Pointer to the starting vertex of the edge

end_vtx Pointer to the ending vertex of the edge. For an unoriented graph, the order of the vertex parameters does not matter.

The function removes the edge connecting two specified vertices. If the vertices are not connected [in that order], the function does nothing.

cvGraphRemoveVtx

Removes a vertex from a graph.

```
int cvGraphRemoveVtx(
           CvGraph* graph,
           int index );
```

graph Graph

vtx_idx Index of the removed vertex

The function removes a vertex from a graph together with all the edges incident to it. The function reports an error if the input vertex does not belong to the graph. The return value is the number of edges deleted, or -1 if the vertex does not belong to the graph.

cvGraphRemoveVtxByPtr

Removes a vertex from a graph by using its pointer.

```
int cvGraphRemoveVtxByPtr(
            CvGraph* graph,
            CvGraphVtx* vtx );
```

graph Graph

vtx Pointer to the removed vertex

The function removes a vertex from the graph by using its pointer together with all the edges incident to it. The function reports an error if the vertex does not belong to the graph. The return value is the number of edges deleted, or -1 if the vertex does not belong to the graph.

cvGraphVtxDegree

Counts the number of edges indicent to the vertex.

```
int cvGraphVtxDegree( const CvGraph* graph, int vtxIdx );
```

graph Graph

vtxIdx Index of the graph vertex

The function returns the number of edges incident to the specified vertex, both incoming and outgoing. To count the edges, the following code is used:

```
CvGraphEdge* edge = vertex->first; int count = 0;
while( edge )
{
    edge = CV_NEXT_GRAPH_EDGE( edge, vertex );
    count++;
}
```

The macro `CV_NEXT_GRAPH_EDGE( edge, vertex )` returns the edge incident to `vertex` that follows after `edge`.

cvGraphVtxDegreeByPtr

Finds an edge in a graph.

```
int cvGraphVtxDegreeByPtr(
           const CvGraph* graph,
           const CvGraphVtx* vtx );
```

graph Graph

vtx Pointer to the graph vertex

The function returns the number of edges incident to the specified vertex, both incoming and outcoming.

cvGraphVtxIdx

Returns the index of a graph vertex.

```
int cvGraphVtxIdx(
           CvGraph* graph,
           CvGraphVtx* vtx );
```

graph Graph

vtx Pointer to the graph vertex

The function returns the index of a graph vertex.

cvInitTreeNodeIterator

Initializes the tree node iterator.

```
void cvInitTreeNodeIterator(
           CvTreeNodeIterator* tree_iterator,
           const void* first,
           int max_level );
```

tree_iterator Tree iterator initialized by the function

first The initial node to start traversing from

max_level The maximal level of the tree (`first` node assumed to be at the first level) to traverse up to. For example, 1 means that only nodes at the same level as `first` should be visited, 2 means that the nodes on the same level as `first` and their direct children should be visited, and so forth.

The function initializes the tree iterator. The tree is traversed in depth-first order.

---

cvInsertNodeIntoTree

Adds a new node to a tree.

```
void cvInsertNodeIntoTree(
           void* node,
           void* parent,
           void* frame );
```

node The inserted node

parent The parent node that is already in the tree

frame The top level node. If `parent` and `frame` are the same, the `v_prev` field of `node` is set to NULL rather than `parent`.

The function adds another node into tree. The function does not allocate any memory, it can only modify links of the tree nodes.

---

cvMakeSeqHeaderForArray

Constructs a sequence header for an array.

```
CvSeq* cvMakeSeqHeaderForArray(
           int seq_type,
           int header_size,
           int elem_size,
```

```
            void* elements,
            int total,
            CvSeq* seq,
            CvSeqBlock* block );
```

seq_type Type of the created sequence

header_size Size of the header of the sequence. Parameter sequence must point to the structure of that size or greater

elem_size Size of the sequence elements

elements Elements that will form a sequence

total Total number of elements in the sequence. The number of array elements must be equal to the value of this parameter.

seq Pointer to the local variable that is used as the sequence header

block Pointer to the local variable that is the header of the single sequence block

The function initializes a sequence header for an array. The sequence header as well as the sequence block are allocated by the user (for example, on stack). No data is copied by the function. The resultant sequence will consists of a single block and have NULL storage pointer; thus, it is possible to read its elements, but the attempts to add elements to the sequence will raise an error in most cases.

---

cvMemStorageAlloc

Allocates a memory buffer in a storage block.

```
void* cvMemStorageAlloc(
            CvMemStorage* storage,
            size_t size );
```

storage Memory storage

size Buffer size

The function allocates a memory buffer in a storage block. The buffer size must not exceed the storage block size, otherwise a runtime error is raised. The buffer address is aligned by `CV_STRUCT_ALIGN=sizeof(double)` (for the moment) bytes.

cvMemStorageAllocString

Allocates a text string in a storage block.

```
CvString cvMemStorageAllocString(CvMemStorage* storage, const char*
ptr, int len=-1);
```

```
typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;
```

storage Memory storage

ptr The string

len Length of the string (not counting the ending NUL) . If the parameter is negative, the function computes the length.

The function creates copy of the string in memory storage. It returns the structure that contains user-passed or computed length of the string and pointer to the copied string.

cvNextGraphItem

Executes one or more steps of the graph traversal procedure.

```
int cvNextGraphItem( CvGraphScanner* scanner );
```

scanner Graph traversal state. It is updated by this function.

The function traverses through the graph until an event of interest to the user (that is, an event, specified in the mask in the cvCreateGraphScanner call) is met or the traversal is completed. In the first case, it returns one of the events listed in the description of the mask parameter above and with the next call it resumes the traversal. In the latter case, it returns CV_GRAPH_OVER (-1). When the event is CV_GRAPH_VERTEX, CV_GRAPH_BACKTRACKING, or CV_GRAPH_NEW_TREE, the currently observed vertex is stored in scanner->vtx. And if the event is edge-related, the edge itself is stored at scanner->edge, the previously visited vertex - at scanner->vtx and the other ending vertex of the edge - at scanner->dst.

cvNextTreeNode

Returns the currently observed node and moves the iterator toward the next node.

```
void* cvNextTreeNode( CvTreeNodeIterator* tree_iterator );
```

tree_iterator Tree iterator initialized by the function

The function returns the currently observed node and then updates the iterator - moving it toward the next node. In other words, the function behavior is similar to the `*p++` expression on a typical C pointer or C++ collection iterator. The function returns NULL if there are no more nodes.

cvPrevTreeNode

Returns the currently observed node and moves the iterator toward the previous node.

```
void* cvPrevTreeNode( CvTreeNodeIterator* tree_iterator );
```

tree_iterator Tree iterator initialized by the function

The function returns the currently observed node and then updates the iterator - moving it toward the previous node. In other words, the function behavior is similar to the `*p--` expression on a typical C pointer or C++ collection iterator. The function returns NULL if there are no more nodes.

cvReleaseGraphScanner

Completes the graph traversal procedure.

```
void cvReleaseGraphScanner( CvGraphScanner** scanner );
```

scanner Double pointer to graph traverser

The function completes the graph traversal procedure and releases the traverser state.

cvReleaseMemStorage

Releases memory storage.

```
void cvReleaseMemStorage( CvMemStorage** storage );
```

storage Pointer to the released storage

The function deallocates all storage memory blocks or returns them to the parent, if any. Then it deallocates the storage header and clears the pointer to the storage. All child storage associated with a given parent storage block must be released before the parent storage block is released.

cvRestoreMemStoragePos

Restores memory storage position.

```
void cvRestoreMemStoragePos(
           CvMemStorage* storage,
           CvMemStoragePos* pos);
```

storage Memory storage

pos New storage top position

The function restores the position of the storage top from the parameter `pos`. This function and the function `cvClearMemStorage` are the only methods to release memory occupied in memory blocks. Note again that there is no way to free memory in the middle of an occupied portion of a storage block.

cvSaveMemStoragePos

Saves memory storage position.

```
void cvSaveMemStoragePos(
           const CvMemStorage* storage,
           CvMemStoragePos* pos);
```

storage Memory storage

pos The output position of the storage top

The function saves the current position of the storage top to the parameter `pos`. The function `cvRestoreMemStoragePos` can further retrieve this position.

---

cvSeqElemIdx

Returns the index of a specific sequence element.

```
int cvSeqElemIdx(
           const CvSeq* seq,
           const void* element,
           CvSeqBlock** block=NULL );
```

seq Sequence

element Pointer to the element within the sequence

block Optional argument. If the pointer is not `NULL`, the address of the sequence block that contains the element is stored in this location.

The function returns the index of a sequence element or a negative number if the element is not found.

---

cvSeqInsert

Inserts an element in the middle of a sequence.

```
char* cvSeqInsert(
           CvSeq* seq,
           int beforeIndex,
           void* element=NULL );
```

seq Sequence

beforeIndex Index before which the element is inserted. Inserting before 0 (the minimal allowed value of the parameter) is equal to cvSeqPushFront and inserting before `seq->total` (the maximal allowed value of the parameter) is equal to cvSeqPush.

element Inserted element

The function shifts the sequence elements from the inserted position to the nearest end of the sequence and copies the `element` content there if the pointer is not NULL. The function returns a pointer to the inserted element.

---

cvSeqInsertSlice

Inserts an array in the middle of a sequence.

```
void cvSeqInsertSlice(
           CvSeq* seq,
           int beforeIndex,
           const CvArr* fromArr );
```

seq Sequence

slice The part of the sequence to remove

fromArr The array to take elements from

The function inserts all `fromArr` array elements at the specified position of the sequence. The array `fromArr` can be a matrix or another sequence.

---

cvSeqInvert

Reverses the order of sequence elements.

```
void cvSeqInvert( CvSeq* seq );
```

seq Sequence

The function reverses the sequence in-place - makes the first element go last, the last element go first and so forth.

cvSeqPop

Removes an element from the end of a sequence.

```
void cvSeqPop(
           CvSeq* seq,
           void* element=NULL );
```

seq Sequence

element Optional parameter . If the pointer is not zero, the function copies the removed element to this location.

The function removes an element from a sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

cvSeqPopFront

Removes an element from the beginning of a sequence.

```
void cvSeqPopFront(

           CvSeq* seq,

           void* element=NULL );
```

seq Sequence

element Optional parameter. If the pointer is not zero, the function copies the removed element to this location.

The function removes an element from the beginning of a sequence. The function reports an error if the sequence is already empty. The function has O(1) complexity.

cvSeqPopMulti

Removes several elements from either end of a sequence.

```
void cvSeqPopMulti(
           CvSeq* seq,
           void* elements,
           int count,
           int in_front=0 );
```

seq Sequence

elements Removed elements

count Number of elements to pop

in_front The flags specifying which end of the modified sequence.

- **CV_BACK** the elements are added to the end of the sequence
- **CV_FRONT** the elements are added to the beginning of the sequence

The function removes several elements from either end of the sequence. If the number of the elements to be removed exceeds the total number of elements in the sequence, the function removes as many elements as possible.

cvSeqPush

Adds an element to the end of a sequence.

```
char* cvSeqPush(
           CvSeq* seq,
           void* element=NULL );
```

seq Sequence

element Added element

The function adds an element to the end of a sequence and returns a pointer to the allocated element. If the input `element` is NULL, the function simply allocates a space for one more element.

The following code demonstrates how to create a new sequence using this function:

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, /* sequence of integer elements */
                          sizeof(CvSeq), /* header size - no extra fields */
                          sizeof(int), /* element size */
                          storage /* the container storage */ );
int i;
for( i = 0; i < 100; i++ )
{
    int* added = (int*)cvSeqPush( seq, &i );
    printf( "%d is added\n", *added );
}

...
/* release memory storage in the end */
cvReleaseMemStorage( &storage );
```

The function has O(1) complexity, but there is a faster method for writing large sequences (see cvStartWriteSeq and related functions).

---

cvSeqPushFront

Adds an element to the beginning of a sequence.

```
char* cvSeqPushFront( CvSeq* seq, void* element=NULL );
```

seq Sequence

element Added element

The function is similar to cvSeqPush but it adds the new element to the beginning of the sequence. The function has O(1) complexity.

---

cvSeqPushMulti

Pushes several elements to either end of a sequence.

```
void cvSeqPushMulti(
           CvSeq* seq,
           void* elements,
           int count,
           int in_front=0 );
```

seq Sequence

elements Added elements

count Number of elements to push

in_front The flags specifying which end of the modified sequence.

- **CV_BACK** the elements are added to the end of the sequence
- **CV_FRONT** the elements are added to the beginning of the sequence

The function adds several elements to either end of a sequence. The elements are added to the sequence in the same order as they are arranged in the input array but they can fall into different sequence blocks.

---

cvSeqRemove

Removes an element from the middle of a sequence.

```
void cvSeqRemove(
           CvSeq* seq,
           int index );
```

seq Sequence

index Index of removed element

The function removes elements with the given index. If the index is out of range the function reports an error. An attempt to remove an element from an empty sequence is a special case of this situation. The function removes an element by shifting the sequence elements between the nearest end of the sequence and the `index`-th position, not counting the latter.

cvSeqRemoveSlice

Removes a sequence slice.

```
void cvSeqRemoveSlice( CvSeq* seq, CvSlice slice );
```

seq Sequence

slice The part of the sequence to remove

The function removes a slice from the sequence.

cvSeqSearch

Searches for an element in a sequence.

```
char* cvSeqSearch( CvSeq* seq, const void* elem, CvCmpFunc func, int
is_sorted, int* elem_idx, void* userdata=NULL );
```

seq The sequence

elem The element to look for

func The comparison function that returns negative, zero or positive value depending on the relationships among the elements (see also cvSeqSort)

is_sorted Whether the sequence is sorted or not

elem_idx Output parameter; index of the found element

userdata The user parameter passed to the compasion function; helps to avoid global variables in some cases

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
```

The function searches for the element in the sequence. If the sequence is sorted, a binary O(log(N)) search is used; otherwise, a simple linear search is used. If the element is not found, the function returns a NULL pointer and the index is set to the number of sequence elements if a linear search is used, or to the smallest index `i, seq(i)>elem`.

cvSeqSlice

Makes a separate header for a sequence slice.

```
CvSeq* cvSeqSlice(
           const CvSeq* seq,
           CvSlice slice,
           CvMemStorage* storage=NULL,
           int copy_data=0 );
```

seq Sequence

slice The part of the sequence to be extracted

storage The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

copy_data The flag that indicates whether to copy the elements of the extracted slice (`copy_data!=0`) or not (`copy_data=0`)

The function creates a sequence that represents the specified slice of the input sequence. The new sequence either shares the elements with the original sequence or has its own copy of the elements. So if one needs to process a part of sequence but the processing function does not have a slice parameter, the required sub-sequence may be extracted using this function.

cvSeqSort

Sorts sequence element using the specified comparison function.

```
void cvSeqSort( CvSeq* seq, CvCmpFunc func, void* userdata=NULL );
```

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
```

seq The sequence to sort

func The comparison function that returns a negative, zero, or positive value depending on the relationships among the elements (see the above declaration and the example below) - a similar function is used by `qsort` from C runline except that in the latter, `userdata` is not used

userdata The user parameter passed to the compasion function; helps to avoid global variables in some cases

The function sorts the sequence in-place using the specified criteria. Below is an example of using this function:

```
/* Sort 2d points in top-to-bottom left-to-right order */
static int cmp_func( const void* _a, const void* _b, void* userdata )
{
    CvPoint* a = (CvPoint*)_a;
    CvPoint* b = (CvPoint*)_b;
    int y_diff = a->y - b->y;
    int x_diff = a->x - b->x;
    return y_diff ? y_diff : x_diff;
}

...

CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq), sizeof(CvPoint), storage );
int i;

for( i = 0; i < 10; i++ )
{
    CvPoint pt;
    pt.x = rand() % 1000;
    pt.y = rand() % 1000;
    cvSeqPush( seq, &pt );
}

cvSeqSort( seq, cmp_func, 0 /* userdata is not used here */ );

/* print out the sorted sequence */
for( i = 0; i < seq->total; i++ )
{
    CvPoint* pt = (CvPoint*)cvSeqElem( seq, i );
    printf( "(%d,%d)\n", pt->x, pt->y );
}

cvReleaseMemStorage( &storage );
```

cvSetAdd

Occupies a node in the set.

```
int cvSetAdd(
            CvSet* setHeader,
            CvSetElem* elem=NULL,
            CvSetElem** inserted_elem=NULL );
```

setHeader Set

elem Optional input argument, an inserted element. If not NULL, the function copies the data to the allocated node (the MSB of the first integer field is cleared after copying).

inserted_elem Optional output argument; the pointer to the allocated cell

The function allocates a new node, optionally copies input element data to it, and returns the pointer and the index to the node. The index value is taken from the lower bits of the `flags` field of the node. The function has O(1) complexity; however, there exists a faster function for allocating set nodes (see cvSetNew).

cvSetNew

Adds an element to a set (fast variant).

```
CvSetElem* cvSetNew( CvSet* setHeader );
```

setHeader Set

The function is an inline lightweight variant of cvSetAdd. It occupies a new node and returns a pointer to it rather than an index.

cvSetRemove

Removes an element from a set.

```
void cvSetRemove(
            CvSet* setHeader,
            int index );
```

setHeader Set

index Index of the removed element

The function removes an element with a specified index from the set. If the node at the specified location is not occupied, the function does nothing. The function has O(1) complexity; however, cvSetRemoveByPtr provides a quicker way to remove a set element if it is located already.

---

cvSetRemoveByPtr

Removes a set element based on its pointer.

```
void cvSetRemoveByPtr(
            CvSet* setHeader,
            void* elem );
```

setHeader Set

elem Removed element

The function is an inline lightweight variant of cvSetRemove that requires an element pointer. The function does not check whether the node is occupied or not - the user should take care of that.

---

cvSetSeqBlockSize

Sets up sequence block size.

```
void cvSetSeqBlockSize(
            CvSeq* seq,
            int deltaElems );
```

seq Sequence

deltaElems Desirable sequence block size for elements

The function affects memory allocation granularity. When the free space in the sequence buffers has run out, the function allocates the space for `deltaElems` sequence elements. If this block immediately follows the one previously allocated, the two blocks are concatenated; otherwise, a new sequence block is created. Therefore, the bigger the parameter is, the lower the possible sequence fragmentation, but the more space in the storage block is wasted. When the sequence is created, the parameter `deltaElems` is set to the default value of about 1K. The function can be called any time after the sequence is created and affects future allocations. The function can modify the passed value of the parameter to meet memory storage constraints.

---

cvSetSeqReaderPos

Moves the reader to the specified position.

```
void cvSetSeqReaderPos(
           CvSeqReader* reader,
           int index,
           int is_relative=0 );
```

reader Reader state

index The destination position. If the positioning mode is used (see the next parameter), the actual position will be `index` mod `reader->seq->total`.

is_relative If it is not zero, then `index` is a relative to the current position

The function moves the read position to an absolute position or relative to the current position.

---

cvStartAppendToSeq

Initializes the process of writing data to a sequence.

```
void cvStartAppendToSeq(
           CvSeq* seq,
           CvSeqWriter* writer );
```

seq Pointer to the sequence

writer Writer state; initialized by the function

The function initializes the process of writing data to a sequence. Written elements are added to the end of the sequence by using the `CV_WRITE_SEQ_ELEM( written_elem, writer )` macro. Note that during the writing process, other operations on the sequence may yield an incorrect result or even corrupt the sequence (see description of cvFlushSeqWriter, which helps to avoid some of these problems).

---

cvStartReadSeq

Initializes the process of sequential reading from a sequence.

```
void cvStartReadSeq(
           const CvSeq* seq,
           CvSeqReader* reader,
           int reverse=0 );
```

seq Sequence

reader Reader state; initialized by the function

reverse Determines the direction of the sequence traversal. If `reverse` is 0, the reader is positioned at the first sequence element; otherwise it is positioned at the last element.

The function initializes the reader state. After that, all the sequence elements from the first one down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM( read_elem, reader )` in the case of forward reading and by using `CV_REV_READ_SEQ_ELEM( read_elem, reader )` in the case of reverse reading. Both macros put the sequence element to `read_elem` and move the reading pointer toward the next element. A circular structure of sequence blocks is used for the reading process, that is, after the last element has been read by the macro `CV_READ_SEQ_ELEM`, the first element is read when the macro is called again. The same applies to `CV_REV_READ_SEQ_ELEM`. There is no function to finish the reading process, since it neither changes the sequence nor creates any temporary buffers. The reader field `ptr` points to the current element of the sequence that is to be read next. The code below demonstrates how to use the sequence writer and reader.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, sizeof(CvSeq), sizeof(int), storage );
```

```
CvSeqWriter writer;
CvSeqReader reader;
int i;

cvStartAppendToSeq( seq, &writer );
for( i = 0; i < 10; i++ )
{
    int val = rand()%100;
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("%d is written\n", val );
}
cvEndWriteSeq( &writer );

cvStartReadSeq( seq, &reader, 0 );
for( i = 0; i < seq->total; i++ )
{
    int val;
#if 1
    CV_READ_SEQ_ELEM( val, reader );
    printf("%d is read\n", val );
#else /* alternative way, that is prefferable if sequence elements are large,
         or their size/type is unknown at compile time */
    printf("%d is read\n", *(int*)reader.ptr );
    CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
#endif
}
...

cvReleaseStorage( &storage );
```

---

cvStartWriteSeq

Creates a new sequence and initializes a writer for it.

```
void cvStartWriteSeq(
           int seq_flags,
           int header_size,
           int elem_size,
           CvMemStorage* storage,
           CvSeqWriter* writer );
```

seq_flags Flags of the created sequence. If the sequence is not passed to any function working with a specific type of sequences, the sequence value may be equal to 0; otherwise the appropriate type must be selected from the list of predefined sequence types.

header_size Size of the sequence header. The parameter value may not be less than `sizeof(CvSeq)`. If a certain type or extension is specified, it must fit within the base type header.

elem_size Size of the sequence elements in bytes; must be consistent with the sequence type. For example, if a sequence of points is created (element type `CV_SEQ_ELTYPE_POINT`), then the parameter `elem_size` must be equal to `sizeof(CvPoint)`.

storage Sequence location

writer Writer state; initialized by the function

The function is a combination of cvCreateSeq and cvStartAppendToSeq. The pointer to the created sequence is stored at `writer->seq` and is also returned by the cvEndWriteSeq function that should be called at the end.

---

cvTreeToNodeSeq

Gathers all node pointers to a single sequence.

```
CvSeq* cvTreeToNodeSeq(
           const void* first,
           int header_size,
           CvMemStorage* storage );
```

first The initial tree node

header_size Header size of the created sequence (sizeof(CvSeq) is the most frequently used value)

storage Container for the sequence

The function puts pointers of all nodes reacheable from `first` into a single sequence. The pointers are written sequentially in the depth-first order.

1.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with `CV_RGB` macro or the cv function) for color images and brightness for grayscale images. For color images the order channel is normally *Blue, Green, Red*, this is what cv, cv and cv expect , so if you form a color using cv, it should look like:

$$\texttt{cvScalar}(blue\_component, green\_component, red\_component[, alpha\_component])$$

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using cv.

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}(x*2^{-shift}, y*2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

---

cvCircle

Draws a circle.

```
void cvCircle(
           CvArr* img,
           CvPoint center,
           int radius,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img Image where the circle is drawn

center Center of the circle

radius Radius of the circle

color Circle color

thickness Thickness of the circle outline if positive, otherwise this indicates that a filled circle is to be drawn

lineType Type of the circle boundary, see Line description

shift Number of fractional bits in the center coordinates and radius value

The function draws a simple or filled circle with a given center and radius.

---

cvClipLine

Clips the line against the image rectangle.

```
int cvClipLine(
           CvSize imgSize,
           CvPoint* pt1,
           CvPoint* pt2 );
```

imgSize Size of the image

pt1 First ending point of the line segment. It is modified by the function.

pt2 Second ending point of the line segment. It is modified by the function.

The function calculates a part of the line segment which is entirely within the image. It returns 0 if the line segment is completely outside the image and 1 otherwise.

---

cvDrawContours

Draws contour outlines or interiors in an image.

```
void cvDrawContours(
           CvArr *img,
           CvSeq* contour,
           CvScalar external_color,
           CvScalar hole_color,
           int max_level,
           int thickness=1,
           int lineType=8 );
```

img Image where the contours are to be drawn. As with any other drawing function, the contours are clipped with the ROI.

contour Pointer to the first contour

external_color Color of the external contours

hole_color Color of internal contours (holes)

max_level Maximal level for drawn contours. If 0, only `contour` is drawn. If 1, the contour and all contours following it on the same level are drawn. If 2, all contours following and all contours one level below the contours are drawn, and so forth. If the value is negative, the function does not draw the contours following after `contour` but draws the child contours of `contour` up to the $|\texttt{max\_level}| - 1$ level.

thickness Thickness of lines the contours are drawn with. If it is negative (For example, =CV_FILLED), the contour interiors are drawn.

lineType Type of the contour segments, see Line description

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$.

Example: Connected component detection via contour functions

```
#include "cv.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    IplImage* src;
    // the first command line parameter must be file name of binary
    // (black-n-white) image
```

```
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* contour = 0;

        cvThreshold( src, src, 1, 255, CV_THRESH_BINARY );
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvFindContours( src, storage, &contour, sizeof(CvContour),
           CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );
        cvZero( dst );

        for( ; contour != 0; contour = contour->h_next )
        {
            CvScalar color = CV_RGB( rand()&255, rand()&255, rand()&255 );
            /* replace CV_FILLED with 1 to see the outlines */
            cvDrawContours( dst, contour, color, color, -1, CV_FILLED, 8 );
        }

        cvNamedWindow( "Components", 1 );
        cvShowImage( "Components", dst );
        cvWaitKey(0);
    }
}
```

---

cvEllipse

Draws a simple or thick elliptic arc or an fills ellipse sector.

```
void cvEllipse(
           CvArr* img,
           CvPoint center,
           CvSize axes,
           double angle,
           double start_angle,
           double end_angle,
           CvScalar color,
           int thickness=1,
           int lineType=8,
```

```
            int shift=0 );
```

img The image

center Center of the ellipse

axes Length of the ellipse axes

angle Rotation angle

start_angle Starting angle of the elliptic arc

end_angle Ending angle of the elliptic arc.

color Ellipse color

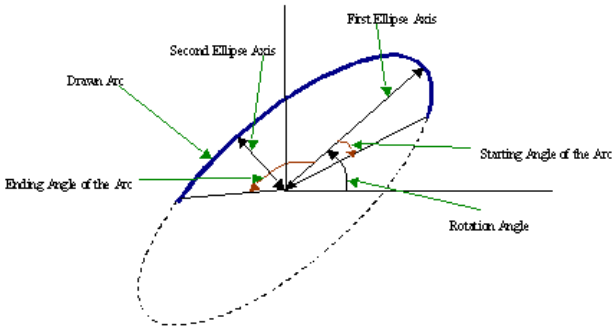
thickness Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the center coordinates and axes' values

The function draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by the ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are given in degrees. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



cvEllipseBox

Draws a simple or thick elliptic arc or fills an ellipse sector.

```
void cvEllipseBox(
           CvArr* img,
           CvBox2D box,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img Image

box The enclosing box of the ellipse drawn

thickness Thickness of the ellipse boundary

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the box vertex coordinates

The function draws a simple or thick ellipse outline, or fills an ellipse. The functions provides a convenient way to draw an ellipse approximating some shape; that is what CamShift and FitEllipse do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

cvFillConvexPoly

Fills a convex polygon.

```
void cvFillConvexPoly(
           CvArr* img,
           CvPoint* pts,
           int npts,
           CvScalar color,
           int lineType=8,
           int shift=0 );
```

img Image

pts Array of pointers to a single polygon

npts Polygon vertex counter

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills a convex polygon's interior. This function is much faster than the function `cvFillPoly` and can fill not only convex polygons but any monotonic polygon, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most.

---

cvFillPoly

Fills a polygon's interior.

```
void cvFillPoly(
           CvArr* img,
           CvPoint** pts,
           int* npts,
           int contours,
           CvScalar color,
           int lineType=8,
           int shift=0 );
```

img Image

pts Array of pointers to polygons

npts Array of polygon vertex counters

contours Number of contours that bind the filled region

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, and so forth.

cvGetTextSize

Retrieves the width and height of a text string.

```
void cvGetTextSize(
           const char* textString,
           const CvFont* font,
           CvSize* textSize,
           int* baseline );
```

font Pointer to the font structure

textString Input string

textSize Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.

baseline y-coordinate of the baseline relative to the bottom-most text point

The function calculates the dimensions of a rectangle to enclose a text string when a specified font is used.

cvInitFont

Initializes font structure.

```
void cvInitFont(
           CvFont* font,
           int fontFace,
           double hscale,
           double vscale,
           double shear=0,
           int thickness=1,
           int lineType=8 );
```

font Pointer to the font structure initialized by the function

fontFace Font name identifier. Only a subset of Hershey fonts http://sources.isc.org/utils/misc/hershey-font.txt are supported now:

- **CV_FONT_HERSHEY_SIMPLEX** normal size sans-serif font
- **CV_FONT_HERSHEY_PLAIN** small size sans-serif font
- **CV_FONT_HERSHEY_DUPLEX** normal size sans-serif font (more complex than CV_FONT_HERSHEY_SIMPLEX)
- **CV_FONT_HERSHEY_COMPLEX** normal size serif font
- **CV_FONT_HERSHEY_TRIPLEX** normal size serif font (more complex than CV_FONT_HERSHEY_COMPLEX)
- **CV_FONT_HERSHEY_COMPLEX_SMALL** smaller version of CV_FONT_HERSHEY_COMPLEX
- **CV_FONT_HERSHEY_SCRIPT_SIMPLEX** hand-writing style font
- **CV_FONT_HERSHEY_SCRIPT_COMPLEX** more complex variant of CV_FONT_HERSHEY_SCRIPT_SIMPLEX

The parameter can be composited from one of the values above and an optional CV_FONT_ITALIC flag, which indicates italic or oblique font.

hscale Horizontal scale. If equal to 1.0f, the characters have the original width depending on the font type. If equal to 0.5f, the characters are of half the original width.

vscale Vertical scale. If equal to 1.0f, the characters have the original height depending on the font type. If equal to 0.5f, the characters are of half the original height.

shear Approximate tangent of the character slope relative to the vertical line. A zero value means a non-italic font, 1.0f means about a 45 degree slope, etc.

thickness Thickness of the text strokes

lineType Type of the strokes, see Line description

The function initializes the font structure that can be passed to text rendering functions.

---

cvInitLineIterator

Initializes the line iterator.

```
int cvInitLineIterator(
           const CvArr* image,
           CvPoint pt1,
           CvPoint pt2,
```

```
            CvLineIterator* line_iterator,
            int connectivity=8,
            int left_to_right=0 );
```

image Image to sample the line from

pt1 First ending point of the line segment

pt2 Second ending point of the line segment

line_iterator Pointer to the line iterator state structure

connectivity The scanned line connectivity, 4 or 8.

left_to_right If ($\texttt{left\_to\_right} = 0$) then the line is scanned in the specified order, from `pt1` to `pt2`. If ($\texttt{left\_to\_right} \neq 0$) the line is scanned from left-most point to right-most.

The function initializes the line iterator and returns the number of pixels between the two end points. Both points must be inside the image. After the iterator has been initialized, all the points on the raster line that connects the two ending points may be retrieved by successive calls of `CV_NEXT_LINE_POINT` point. The points on the line are calculated one by one using a 4-connected or 8-connected Bresenham algorithm.

```
CvScalar sum_line_pixels( IplImage* image, CvPoint pt1, CvPoint pt2 )
{
    CvLineIterator iterator;
    int blue_sum = 0, green_sum = 0, red_sum = 0;
    int count = cvInitLineIterator( image, pt1, pt2, &iterator, 8, 0 );

    for( int i = 0; i < count; i++ ){
        blue_sum += iterator.ptr[0];
        green_sum += iterator.ptr[1];
        red_sum += iterator.ptr[2];
        CV_NEXT_LINE_POINT(iterator);

        /* print the pixel coordinates: demonstrates how to calculate the
                                                        coordinates */
        {
        int offset, x, y;
        /* assume that ROI is not set, otherwise need to take it
                                                into account. */
        offset = iterator.ptr - (uchar*)(image->imageData);
```

```
        y = offset/image->widthStep;
        x = (offset - y*image->widthStep)/(3*sizeof(uchar)
                                        /* size of pixel */);
        printf("(%d,%d)\n", x, y );
        }
    }
    return cvScalar( blue_sum, green_sum, red_sum );
}
```

---

cvLine

Draws a line segment connecting two points.

```
void cvLine(
           CvArr* img,
           CvPoint pt1,
           CvPoint pt2,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img The image

pt1 First point of the line segment

pt2 Second point of the line segment

color Line color

thickness Line thickness

lineType Type of the line:

- **8** (or omitted) 8-connected line.
- **4** 4-connected line.
- **CV_AA** antialiased line.

shift Number of fractional bits in the point coordinates

The function draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB( r, g, b )`.

---

cvPolyLine

Draws simple or thick polygons.

```
void cvPolyLine(
           CvArr* img,
           CvPoint** pts,
           int* npts,
           int contours,
           int is_closed,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

pts Array of pointers to polygons

npts Array of polygon vertex counters

contours Number of contours that bind the filled region

img Image

is_closed Indicates whether the polylines must be drawn closed. If closed, the function draws the line from the last vertex of every contour to the first vertex.

color Polyline color

thickness Thickness of the polyline edges

lineType Type of the line segments, see Line description

shift Number of fractional bits in the vertex coordinates

The function draws single or multiple polygonal curves.

cvPutText

Draws a text string.

```
void cvPutText(
           CvArr* img,
           const char* text,
           CvPoint org,
           const CvFont* font,
           CvScalar color );
```

img Input image

text String to print

org Coordinates of the bottom-left corner of the first letter

font Pointer to the font structure

color Text color

The function renders the text in the image with the specified font and color. The printed text is clipped by the ROI rectangle. Symbols that do not belong to the specified font are replaced with the symbol for a rectangle.

cvRectangle

Draws a simple, thick, or filled rectangle.

```
void cvRectangle(
           CvArr* img,
           CvPoint pt1,
           CvPoint pt2,
           CvScalar color,
           int thickness=1,
           int lineType=8,
           int shift=0 );
```

img Image

pt1 One of the rectangle's vertices

pt2 Opposite rectangle vertex

color Line color (RGB) or brightness (grayscale image)

thickness Thickness of lines that make up the rectangle. Negative values, e.g., CV_FILLED, cause the function to draw a filled rectangle.

lineType Type of the line, see Line description

shift Number of fractional bits in the point coordinates

The function draws a rectangle with two opposite corners `pt1` and `pt2`.

CV_RGB

Constructs a color value.

```
#define CV_RGB( r, g, b ) cvScalar( (b), (g), (r) )
```

red Red component

grn Green component

blu Blue component

1.5 XML/YAML Persistence

CvFileStorage

File Storage.

```
typedef struct CvFileStorage
{
    ...       // hidden fields
} CvFileStorage;
```

The structure CvFileStorage is a ”black box” representation of the file storage associated with a file on disk. Several functions that are described below take `CvFileStorage` as inputs and allow theuser to save or to load hierarchical collections that consist of scalar values, standard CXCore objects (such as matrices, sequences, graphs), and user-defined objects.

CXCore can read and write data in XML (http://www.w3c.org/XML) or YAML (http://www.yaml.org) formats. Below is an example of 3×3 floating-point identity matrix `A`, stored in XML and YAML files using CXCore functions:

XML:

```
<?xml version="1.0">
<opencv_storage>
<A type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>f</dt>
  <data>1. 0. 0. 0. 1. 0. 0. 0. 1.</data>
</A>
</opencv_storage>
```

YAML:

```
%YAML:1.0
A: !!opencv-matrix
  rows: 3
  cols: 3
  dt: f
  data: [ 1., 0., 0., 0., 1., 0., 0., 0., 1.]
```

As it can be seen from the examples, XML uses nested tags to represent hierarchy, while YAML uses indentation for that purpose (similar to the Python programming language).

The same CXCore functions can read and write data in both formats; the particular format is determined by the extension of the opened file, .xml for XML files and .yml or .yaml for YAML.

CvFileNode

File Storage Node.

```
/* file node type */
#define CV_NODE_NONE        0
#define CV_NODE_INT         1
#define CV_NODE_INTEGER     CV_NODE_INT
```

```
#define CV_NODE_REAL        2
#define CV_NODE_FLOAT       CV_NODE_REAL
#define CV_NODE_STR         3
#define CV_NODE_STRING      CV_NODE_STR
#define CV_NODE_REF         4 /* not used */
#define CV_NODE_SEQ         5
#define CV_NODE_MAP         6
#define CV_NODE_TYPE_MASK   7

/* optional flags */
#define CV_NODE_USER        16
#define CV_NODE_EMPTY       32
#define CV_NODE_NAMED       64

#define CV_NODE_TYPE(tag)  ((tag) & CV_NODE_TYPE_MASK)

#define CV_NODE_IS_INT(tag)         (CV_NODE_TYPE(tag) == CV_NODE_INT)
#define CV_NODE_IS_REAL(tag)        (CV_NODE_TYPE(tag) == CV_NODE_REAL)
#define CV_NODE_IS_STRING(tag)      (CV_NODE_TYPE(tag) == CV_NODE_STRING)
#define CV_NODE_IS_SEQ(tag)         (CV_NODE_TYPE(tag) == CV_NODE_SEQ)
#define CV_NODE_IS_MAP(tag)         (CV_NODE_TYPE(tag) == CV_NODE_MAP)
#define CV_NODE_IS_COLLECTION(tag)  (CV_NODE_TYPE(tag) >= CV_NODE_SEQ)
#define CV_NODE_IS_FLOW(tag)        (((tag) & CV_NODE_FLOW) != 0)
#define CV_NODE_IS_EMPTY(tag)       (((tag) & CV_NODE_EMPTY) != 0)
#define CV_NODE_IS_USER(tag)        (((tag) & CV_NODE_USER) != 0)
#define CV_NODE_HAS_NAME(tag)       (((tag) & CV_NODE_NAMED) != 0)

#define CV_NODE_SEQ_SIMPLE 256
#define CV_NODE_SEQ_IS_SIMPLE(seq) (((seq)->flags & CV_NODE_SEQ_SIMPLE) != 0)

typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;

/* all the keys (names) of elements in the readed file storage
   are stored in the hash to speed up the lookup operations */
typedef struct CvStringHashNode
{
    unsigned hashval;
    CvString str;
    struct CvStringHashNode* next;
}
```

```
CvStringHashNode;

/* basic element of the file storage - scalar or collection */
typedef struct CvFileNode
{
    int tag;
    struct CvTypeInfo* info; /* type information
            (only for user-defined object, for others it is 0) */
    union
    {
        double f; /* scalar floating-point number */
        int i;    /* scalar integer number */
        CvString str; /* text string */
        CvSeq* seq; /* sequence (ordered collection of file nodes) */
        struct CvMap* map; /* map (collection of named file nodes) */
    } data;
}
CvFileNode;
```

The structure is used only for retrieving data from file storage (i.e., for loading data from the file). When data is written to a file, it is done sequentially, with minimal buffering. No data is stored in the file storage.

In opposite, when data is read from a file, the whole file is parsed and represented in memory as a tree. Every node of the tree is represented by CvFileNode . The type of file node `N` can be retrieved as `CV_NODE_TYPE(N->tag)`. Some file nodes (leaves) are scalars: text strings, integers, or floating-point numbers. Other file nodes are collections of file nodes, which can be scalars or collections in their turn. There are two types of collections: sequences and maps (we use YAML notation, however, the same is true for XML streams). Sequences (do not mix them with CvSeq) are ordered collections of unnamed file nodes; maps are unordered collections of named file nodes. Thus, elements of sequences are accessed by index (GetSeqElem), while elements of maps are accessed by name (GetFileNodeByName). The table below describes the different types of file nodes:

Type	`CV_NODE_TYPE(node->tag)`	Value
Integer	`CV_NODE_INT`	`node->data.i`
Floating-point	`CV_NODE_REAL`	`node->data.f`
Text string	`CV_NODE_STR`	`node->data.str.ptr`
Sequence	`CV_NODE_SEQ`	`node->data.seq`
Map	`CV_NODE_MAP`	`node->data.map` (see below)

There is no need to access the `map` field directly (by the way, `CvMap` is a hidden structure). The elements of the map can be retrieved with the GetFileNodeByName function that takes a pointer to the ”map” file node.

A user (custom) object is an instance of either one of the standard CxCore types, such as CvMat , CvSeq etc., or any type registered with RegisterTypeInfo . Such an object is initially represented in a file as a map (as shown in XML and YAML example files above) after the file storage has been opened and parsed. Then the object can be decoded (coverted to native representation) by request - when a user calls the Read or ReadByName functions.

CvAttrList

List of attributes.

```
typedef struct CvAttrList
{
    const char** attr; /* NULL-terminated array of (attribute\_name,attribute\_value) pairs
    struct CvAttrList* next; /* pointer to next chunk of the attributes list */
}
CvAttrList;

/* initializes CvAttrList structure */
inline CvAttrList cvAttrList( const char** attr=NULL, CvAttrList* next=NULL );

/* returns attribute value or 0 (NULL) if there is no such attribute */
const char* cvAttrValue( const CvAttrList* attr, const char* attr\_name );
```

In the current implementation, attributes are used to pass extra parameters when writing user objects (see Write). XML attributes inside tags are not supported, aside from the object type specification (`type_id` attribute).

CvTypeInfo

Type information.

```
typedef int (CV_CDECL *CvIsInstanceFunc)( const void* structPtr );
typedef void (CV_CDECL *CvReleaseFunc)( void** structDblPtr );
typedef void* (CV_CDECL *CvReadFunc)( CvFileStorage* storage, CvFileNode* node );
typedef void (CV_CDECL *CvWriteFunc)( CvFileStorage* storage,
                                      const char* name,
                                      const void* structPtr,
                                      CvAttrList attributes );
typedef void* (CV_CDECL *CvCloneFunc)( const void* structPtr );

typedef struct CvTypeInfo
{
    int flags; /* not used */
    int header_size; /* sizeof(CvTypeInfo) */
```

```
    struct CvTypeInfo* prev; /* previous registered type in the list */
    struct CvTypeInfo* next; /* next registered type in the list */
    const char* type_name; /* type name, written to file storage */

    /* methods */
    CvIsInstanceFunc is_instance; /* checks if the passed object belongs to the type */
    CvReleaseFunc release; /* releases object (memory etc.) */
    CvReadFunc read; /* reads object from file storage */
    CvWriteFunc write; /* writes object to file storage */
    CvCloneFunc clone; /* creates a copy of the object */
}
CvTypeInfo;
```

The structure CvTypeInfo contains information about one of the standard or user-defined types. Instances of the type may or may not contain a pointer to the corresponding CvTypeInfo structure. In any case, there is a way to find the type info structure for a given object using the TypeOf function. Aternatively, type info can be found by type name using FindType , which is used when an object is read from file storage. The user can register a new type with RegisterType that adds the type information structure into the beginning of the type list. Thus, it is possible to create specialized types from generic standard types and override the basic methods.

---

cvClone

Makes a clone of an object.

```
void* cvClone( const void* structPtr );
```

structPtr The object to clone

The function finds the type of a given object and calls `clone` with the passed object.

---

cvEndWriteStruct

Ends the writing of a structure.

```
void cvEndWriteStruct(CvFileStorage* fs);
```

fs File storage

The function finishes the currently written structure.

cvFindType

Finds a type by its name.

```
CvTypeInfo* cvFindType(const char* typeName);
```

typeName Type name

The function finds a registered type by its name. It returns NULL if there is no type with the specified name.

cvFirstType

Returns the beginning of a type list.

```
CvTypeInfo* cvFirstType(void);
```

The function returns the first type in the list of registered types. Navigation through the list can be done via the `prev` and `next` fields of the CvTypeInfo structure.

cvGetFileNode

Finds a node in a map or file storage.

```
CvFileNode* cvGetFileNode(
           CvFileStorage* fs,
           CvFileNode* map,
           const CvStringHashNode* key,
           int createMissing=0 );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node. If both `map` and `key` are NULLs, the function returns the root file node - a map that contains top-level nodes.

key Unique pointer to the node name, retrieved with GetHashedKey

createMissing Flag that specifies whether an absent node should be added to the map

The function finds a file node. It is a faster version of GetFileNodeByName (see GetHashedKey discussion). Also, the function can insert a new node, if it is not in the map yet.

---

cvGetFileNodeByName

Finds a node in a map or file storage.

```
CvFileNode* cvGetFileNodeByName(
            const CvFileStorage* fs,
            const CvFileNode* map,
            const char* name);
```

fs File storage

map The parent map. If it is NULL, the function searches in all the top-level nodes (streams), starting with the first one.

name The file node name

The function finds a file node by `name`. The node is searched either in `map` or, if the pointer is NULL, among the top-level file storage nodes. Using this function for maps and GetSeqElem (or sequence reader) for sequences, it is possible to nagivate through the file storage. To speed up multiple queries for a certain key (e.g., in the case of an array of structures) one may use a combination of GetHashedKey and GetFileNode .

---

cvGetFileNodeName

Returns the name of a file node.

```
const char* cvGetFileNodeName( const CvFileNode* node );
```

node File node

The function returns the name of a file node or NULL, if the file node does not have a name or if `node` is `NULL`.

cvGetHashedKey

Returns a unique pointer for a given name.

```
CvStringHashNode* cvGetHashedKey(
            CvFileStorage* fs,
            const char* name,
            int len=-1,
            int createMissing=0 );
```

fs File storage

name Literal node name

len Length of the name (if it is known apriori), or -1 if it needs to be calculated

createMissing Flag that specifies, whether an absent key should be added into the hash table

The function returns a unique pointer for each particular file node name. This pointer can be then passed to the GetFileNode function that is faster than GetFileNodeByName because it compares text strings by comparing pointers rather than the strings' content.

Consider the following example where an array of points is encoded as a sequence of 2-entry maps:

```
%YAML:1.0
points:
  - { x: 10, y: 10 }
  - { x: 20, y: 20 }
  - { x: 30, y: 30 }
  # ...
```

Then, it is possible to get hashed "x" and "y" pointers to speed up decoding of the points.

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvFileStorage* fs = cvOpenFileStorage( "points.yml", 0, CV\_STORAGE\_READ );
    CvStringHashNode* x\_key = cvGetHashedNode( fs, "x", -1, 1 );
    CvStringHashNode* y\_key = cvGetHashedNode( fs, "y", -1, 1 );
    CvFileNode* points = cvGetFileNodeByName( fs, 0, "points" );
```

```
    if( CV\_NODE\_IS\_SEQ(points->tag) )
    {
        CvSeq* seq = points->data.seq;
        int i, total = seq->total;
        CvSeqReader reader;
        cvStartReadSeq( seq, &reader, 0 );
        for( i = 0; i < total; i++ )
        {
            CvFileNode* pt = (CvFileNode*)reader.ptr;
#if 1 /* faster variant */
            CvFileNode* xnode = cvGetFileNode( fs, pt, x\_key, 0 );
            CvFileNode* ynode = cvGetFileNode( fs, pt, y\_key, 0 );
            assert( xnode && CV\_NODE\_IS\_INT(xnode->tag) &&
                    ynode && CV\_NODE\_IS\_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#elif 1 /* slower variant; does not use x\_key & y\_key */
            CvFileNode* xnode = cvGetFileNodeByName( fs, pt, "x" );
            CvFileNode* ynode = cvGetFileNodeByName( fs, pt, "y" );
            assert( xnode && CV\_NODE\_IS\_INT(xnode->tag) &&
                    ynode && CV\_NODE\_IS\_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#else /* the slowest yet the easiest to use variant */
            int x = cvReadIntByName( fs, pt, "x", 0 /* default value */ );
            int y = cvReadIntByName( fs, pt, "y", 0 /* default value */ );
#endif
            CV\_NEXT\_SEQ\_ELEM( seq->elem\_size, reader );
            printf("%d: (%d, %d)\n", i, x, y );
        }
    }
    cvReleaseFileStorage( &fs );
    return 0;
}
```

Please note that whatever method of accessing a map you are using, it is still much slower than using plain sequences; for example, in the above example, it is more efficient to encode the points as pairs of integers in a single numeric sequence.

---

cvGetRootFileNode

Retrieves one of the top-level nodes of the file storage.

```
CvFileNode* cvGetRootFileNode(
            const CvFileStorage* fs,
            int stream_index=0 );
```

fs File storage

stream_index Zero-based index of the stream. See StartNextStream . In most cases, there is only one stream in the file; however, there can be several.

The function returns one of the top-level file nodes. The top-level nodes do not have a name, they correspond to the streams that are stored one after another in the file storage. If the index is out of range, the function returns a NULL pointer, so all the top-level nodes may be iterated by subsequent calls to the function with `stream_index=0,1,...`, until the NULL pointer is returned. This function may be used as a base for recursive traversal of the file storage.

---

cvLoad

Loads an object from a file.

```
void* cvLoad(
            const char* filename,
            CvMemStorage* storage=NULL,
            const char* name=NULL,
            const char** realName=NULL );
```

filename File name

storage Memory storage for dynamic structures, such as CvSeq or CvGraph . It is not used for matrices or images.

name Optional object name. If it is NULL, the first top-level object in the storage will be loaded.

realName Optional output parameter that will contain the name of the loaded object (useful if `name=NULL`)

The function loads an object from a file. It provides a simple interface to cvRead. After the object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as a sequence, contour, or graph, one should pass a valid memory storage destination to the function.

cvOpenFileStorage

Opens file storage for reading or writing data.

```
CvFileStorage* cvOpenFileStorage(
            const char* filename,
            CvMemStorage* memstorage,
            int flags);
```

filename Name of the file associated with the storage

memstorage Memory storage used for temporary data and for storing dynamic structures, such as CvSeq or CvGraph . If it is NULL, a temporary memory storage is created and used.

flags Can be one of the following:

CV_STORAGE_READ the storage is open for reading

CV_STORAGE_WRITE the storage is open for writing

The function opens file storage for reading or writing data. In the latter case, a new file is created or an existing file is rewritten. The type of the read or written file is determined by the filename extension: `.xml` for `XML` and `.yml` or `.yaml` for `YAML`. The function returns a pointer to the CvFileStorage structure.

cvRead

Decodes an object and returns a pointer to it.

```
void* cvRead(
            CvFileStorage* fs,
            CvFileNode* node,
            CvAttrList* attributes=NULL );
```

fs File storage

node The root object node

attributes Unused parameter

The function decodes a user object (creates an object in a native representation from the file storage subtree) and returns it. The object to be decoded must be an instance of a registered type that supports the `read` method (see CvTypeInfo). The type of the object is determined by the type name that is encoded in the file. If the object is a dynamic structure, it is created either in memory storage and passed to OpenFileStorage or, if a NULL pointer was passed, in temporary memory storage, which is released when ReleaseFileStorage is called. Otherwise, if the object is not a dynamic structure, it is created in a heap and should be released with a specialized function or by using the generic Release .

---

cvReadByName

Finds an object by name and decodes it.

```
void* cvReadByName(
           CvFileStorage* fs,
           const CvFileNode* map,
           const char* name,
           CvAttrList* attributes=NULL );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

attributes Unused parameter

The function is a simple superposition of GetFileNodeByName and Read .

---

cvReadInt

Retrieves an integer value from a file node.

```
int cvReadInt(
           const CvFileNode* node,
           int defaultValue=0 );
```

node File node

defaultValue The value that is returned if `node` is NULL

The function returns an integer that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after GetFileNode without checking for a NULL pointer). If the file node has type `CV_NODE_INT`, then `node->data.i` is returned. If the file node has type `CV_NODE_REAL`, then `node->data.f` is converted to an integer and returned. Otherwise the result is not determined.

---

cvReadIntByName

Finds a file node and returns its value.

```
int cvReadIntByName(
           const CvFileStorage* fs,
           const CvFileNode* map,
           const char* name,
           int defaultValue=0 );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

defaultValue The value that is returned if the file node is not found

The function is a simple superposition of GetFileNodeByName and ReadInt .

---

cvReadRawData

Reads multiple numbers.

```
void cvReadRawData(
           const CvFileStorage* fs,
           const CvFileNode* src,
           void* dst,
           const char* dt);
```

fs File storage

src The file node (a sequence) to read numbers from

dst Pointer to the destination array

dt Specification of each array element. It has the same format as in WriteRawData .

The function reads elements from a file node that represents a sequence of scalars.

---

cvReadRawDataSlice

Initializes file node sequence reader.

```
void cvReadRawDataSlice(
           const CvFileStorage* fs,
           CvSeqReader* reader,
           int count,
           void* dst,
           const char* dt );
```

fs File storage

reader The sequence reader. Initialize it with StartReadRawData .

count The number of elements to read

dst Pointer to the destination array

dt Specification of each array element. It has the same format as in WriteRawData .

The function reads one or more elements from the file node, representing a sequence, to a user-specified array. The total number of read sequence elements is a product of `total` and the number of components in each array element. For example, if dt=`2if`, the function will read $\texttt{total} \times 3$ sequence elements. As with any sequence, some parts of the file node sequence may be skipped or read repeatedly by repositioning the reader using SetSeqReaderPos .

---

cvReadReal

Retrieves a floating-point value from a file node.

```
double cvReadReal(
           const CvFileNode* node,
           double defaultValue=0. );
```

node File node

defaultValue The value that is returned if `node` is NULL

The function returns a floating-point value that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after GetFileNode without checking for a NULL pointer). If the file node has type `CV_NODE_REAL`, then `node->data.f` is returned. If the file node has type `CV_NODE_INT`, then `node->data.f` is converted to floating-point and returned. Otherwise the result is not determined.

---

cvReadRealByName

Finds a file node and returns its value.

```
double cvReadRealByName(
           const CvFileStorage* fs,
           const CvFileNode* map,
           const char* name,
           double defaultValue=0.);
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

defaultValue The value that is returned if the file node is not found

The function is a simple superposition of GetFileNodeByName and ReadReal .

---

cvReadString

Retrieves a text string from a file node.

```
const char* cvReadString(
            const CvFileNode* node,
            const char* defaultValue=NULL );
```

node File node

defaultValue The value that is returned if `node` is NULL

The function returns a text string that is represented by the file node. If the file node is NULL, the `defaultValue` is returned (thus, it is convenient to call the function right after GetFileNode without checking for a NULL pointer). If the file node has type `CV_NODE_STR`, then `node->data.str.ptr` is returned. Otherwise the result is not determined.

---

cvReadStringByName

Finds a file node by its name and returns its value.

```
const char* cvReadStringByName(
            const CvFileStorage* fs,
            const CvFileNode* map,
            const char* name,
            const char* defaultValue=NULL );
```

fs File storage

map The parent map. If it is NULL, the function searches a top-level node.

name The node name

defaultValue The value that is returned if the file node is not found

The function is a simple superposition of GetFileNodeByName and ReadString .

---

cvRegisterType

Registers a new type.

```
void cvRegisterType(const CvTypeInfo* info);
```

info Type info structure

The function registers a new type, which is described by `info`. The function creates a copy of the structure, so the user should delete it after calling the function.

---

cvRelease

Releases an object.

```
void cvRelease( void** structPtr );
```

structPtr Double pointer to the object

The function finds the type of a given object and calls `release` with the double pointer.

---

cvReleaseFileStorage

Releases file storage.

```
void cvReleaseFileStorage(CvFileStorage** fs);
```

fs Double pointer to the released file storage

The function closes the file associated with the storage and releases all the temporary structures. It must be called after all I/O operations with the storage are finished.

---

cvSave

Saves an object to a file.

```
void cvSave(
           const char* filename,
           const void* structPtr,
           const char* name=NULL,
           const char* comment=NULL,
           CvAttrList attributes=cvAttrList());
```

filename File name

structPtr Object to save

name Optional object name. If it is NULL, the name will be formed from `filename`.

comment Optional comment to put in the beginning of the file

attributes Optional attributes passed to Write

The function saves an object to a file. It provides a simple interface to Write .

---

cvStartNextStream

Starts the next stream.

```
void cvStartNextStream(CvFileStorage* fs);
```

fs File storage

The function starts the next stream in file storage. Both YAML and XML support multiple "streams." This is useful for concatenating files or for resuming the writing process.

---

cvStartReadRawData

Initializes the file node sequence reader.

```
void cvStartReadRawData(
           const CvFileStorage* fs,
           const CvFileNode* src,
           CvSeqReader* reader);
```

fs File storage

src The file node (a sequence) to read numbers from

reader Pointer to the sequence reader

The function initializes the sequence reader to read data from a file node. The initialized reader can be then passed to ReadRawDataSlice .

---

cvStartWriteStruct

Starts writing a new structure.

```
void cvStartWriteStruct( CvFileStorage* fs,
           const char* name,
           int struct_flags,
           const char* typeName=NULL,
           CvAttrList attributes=cvAttrList(
           ));
```

fs File storage

name Name of the written structure. The structure can be accessed by this name when the storage is read.

struct_flags A combination one of the following values:

- **CV_NODE_SEQ** the written structure is a sequence (see discussion of CvFileStorage), that is, its elements do not have a name.
- **CV_NODE_MAP** the written structure is a map (see discussion of CvFileStorage), that is, all its elements have names.

One and only one of the two above flags must be specified

CV_NODE_FLOW the optional flag that makes sense only for YAML streams. It means that the structure is written as a flow (not as a block), which is more compact. It is recommended to use this flag for structures or arrays whose elements are all scalars.

typeName Optional parameter - the object type name. In case of XML it is written as a `type_id` attribute of the structure opening tag. In the case of YAML it is written after a colon following

the structure name (see the example in CvFileStorage description). Mainly it is used with user objects. When the storage is read, the encoded type name is used to determine the object type (see CvTypeInfo and FindTypeInfo).

attributes This parameter is not used in the current implementation

The function starts writing a compound structure (collection) that can be a sequence or a map. After all the structure fields, which can be scalars or structures, are written, EndWriteStruct should be called. The function can be used to group some objects or to implement the `write` function for a some user object (see CvTypeInfo).

---

cvTypeOf

Returns the type of an object.

```
CvTypeInfo* cvTypeOf( const void* structPtr );
```

structPtr The object pointer

The function finds the type of a given object. It iterates through the list of registered types and calls the `is_instance` function/method for every type info structure with that object until one of them returns non-zero or until the whole list has been traversed. In the latter case, the function returns NULL.

---

cvUnregisterType

Unregisters the type.

```
void cvUnregisterType( const char* typeName );
```

typeName Name of an unregistered type

The function unregisters a type with a specified name. If the name is unknown, it is possible to locate the type info by an instance of the type using TypeOf or by iterating the type list, starting from FirstType , and then calling `cvUnregisterType(info->typeName)`.

cvWrite

Writes a user object.

```
void cvWrite( CvFileStorage* fs,
              const char* name,
              const void* ptr,
              CvAttrList attributes=cvAttrList(
              ) );
```

fs File storage

name Name of the written object. Should be NULL if and only if the parent structure is a sequence.

ptr Pointer to the object

attributes The attributes of the object. They are specific for each particular type (see the dicsussion below).

The function writes an object to file storage. First, the appropriate type info is found using TypeOf . Then, the `write` method associated with the type info is called.

Attributes are used to customize the writing procedure. The standard types support the following attributes (all the `*dt` attributes have the same format as in WriteRawData):

1. CvSeq

   header_dt description of user fields of the sequence header that follow CvSeq, or CvChain (if the sequence is a Freeman chain) or CvContour (if the sequence is a contour or point sequence)

   dt description of the sequence elements.

   recursive if the attribute is present and is not equal to ”0” or ”false”, the whole tree of sequences (contours) is stored.

2. Cvgraph

   header_dt description of user fields of the graph header that follows CvGraph;

   vertex_dt description of user fields of graph vertices

   edge_dt description of user fields of graph edges (note that the edge weight is always written, so there is no need to specify it explicitly)

Below is the code that creates the YAML file shown in the `CvFileStorage` description:

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvMat* mat = cvCreateMat( 3, 3, CV\_32F );
    CvFileStorage* fs = cvOpenFileStorage( "example.yml", 0, CV\_STORAGE\_WRITE );

    cvSetIdentity( mat );
    cvWrite( fs, "A", mat, cvAttrList(0,0) );

    cvReleaseFileStorage( &fs );
    cvReleaseMat( &mat );
    return 0;
}
```

---

cvWriteComment

Writes a comment.

```
void cvWriteComment(
           CvFileStorage* fs,
           const char* comment,
           int eolComment);
```

fs File storage

comment The written comment, single-line or multi-line

eolComment If non-zero, the function tries to put the comment at the end of current line. If the flag is zero, if the comment is multi-line, or if it does not fit at the end of the current line, the comment starts a new line.

The function writes a comment into file storage. The comments are skipped when the storage is read, so they may be used only for debugging or descriptive purposes.

---

cvWriteFileNode

Writes a file node to another file storage.

```
void cvWriteFileNode(
           CvFileStorage* fs,
           const char* new_node_name,
           const CvFileNode* node,
           int embed );
```

fs Destination file storage

new_file_node New name of the file node in the destination file storage. To keep the existing name, use cvGetFileNodeName

node The written node

embed If the written node is a collection and this parameter is not zero, no extra level of hiararchy is created. Instead, all the elements of `node` are written into the currently written structure. Of course, map elements may be written only to a map, and sequence elements may be written only to a sequence.

The function writes a copy of a file node to file storage. Possible applications of the function are merging several file storages into one and conversion between XML and YAML formats.

---

cvWriteInt

Writes an integer value.

```
void cvWriteInt(
           CvFileStorage* fs,
           const char* name,
           int value);
```

fs File storage

name Name of the written value. Should be NULL if and only if the parent structure is a sequence.

value The written value

The function writes a single integer value (with or without a name) to the file storage.

cvWriteRawData

Writes multiple numbers.

```
void cvWriteRawData(
           CvFileStorage* fs,
           const void* src,
           int len,
           const char* dt );
```

fs File storage

src Pointer to the written array

len Number of the array elements to write

dt Specification of each array element that has the following format `([count]{'u'|'c'|'w'|'s'|'i'|'f'|'d'})...` where the characters correspond to fundamental C types:

- **u** 8-bit unsigned number
- **c** 8-bit signed number
- **w** 16-bit unsigned number
- **s** 16-bit signed number
- **i** 32-bit signed number
- **f** single precision floating-point number
- **d** double precision floating-point number
- **r** pointer, 32 lower bits of which are written as a signed integer. The type can be used to store structures with links between the elements. `count` is the optional counter of values of a given type. For example, `2if` means that each array element is a structure of 2 integers, followed by a single-precision floating-point number. The equivalent notations of the above specification are '`iif`', '`2i1f`' and so forth. Other examples: `u` means that the array consists of bytes, and `2d` means the array consists of pairs of doubles.

The function writes an array, whose elements consist of single or multiple numbers. The function call can be replaced with a loop containing a few WriteInt and WriteReal calls, but a single call is more efficient. Note that because none of the elements have a name, they should be written to a sequence rather than a map.

cvWriteReal

Writes a floating-point value.

```
void cvWriteReal(
           CvFileStorage* fs,
           const char* name,
           double value );
```

fs File storage

name Name of the written value. Should be NULL if and only if the parent structure is a sequence.

value The written value

The function writes a single floating-point value (with or without a name) to file storage. Special values are encoded as follows: NaN (Not A Number) as .NaN, $\pm\infty$ as +.Inf (-.Inf).

The following example shows how to use the low-level writing functions to store custom structures, such as termination criteria, without registering a new type.

```
void write_termcriteria( CvFileStorage* fs, const char* struct_name,
                         CvTermCriteria* termcrit )
{
    cvStartWriteStruct( fs, struct_name, CV_NODE_MAP, NULL, cvAttrList(0,0));
    cvWriteComment( fs, "termination criteria", 1 ); // just a description
    if( termcrit->type & CV_TERMCRIT_ITER )
        cvWriteInteger( fs, "max_iterations", termcrit->max_iter );
    if( termcrit->type & CV_TERMCRIT_EPS )
        cvWriteReal( fs, "accuracy", termcrit->epsilon );
    cvEndWriteStruct( fs );
}
```

cvWriteString

Writes a text string.

```
void cvWriteString(
           CvFileStorage* fs,
           const char* name,
```

```
            const char* str,
            int quote=0 );
```

fs File storage

name Name of the written string . Should be NULL if and only if the parent structure is a sequence.

str The written text string

quote If non-zero, the written string is put in quotes, regardless of whether they are required. Otherwise, if the flag is zero, quotes are used only when they are required (e.g. when the string starts with a digit or contains spaces).

The function writes a text string to file storage.

1.6 Clustering and Search in Multi-Dimensional Spaces

---

cvKMeans2

Splits set of vectors by a given number of clusters.

```
int cvKMeans2(const CvArr* samples, int nclusters,
            CvArr* labels, CvTermCriteria termcrit,
            int attempts=1, CvRNG* rng=0,
            int flags=0, CvArr* centers=0,
            double* compactness=0);
```

samples Floating-point matrix of input samples, one row per sample

nclusters Number of clusters to split the set by

labels Output integer vector storing cluster indices for every sample

termcrit Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

attempts How many times the algorithm is executed using different initial labelings. The algorithm returns labels that yield the best compactness (see the last function parameter)

rng Optional external random number generator; can be used to fully control the function behaviour

flags Can be 0 or CV_KMEANS_USE_INITIAL_LABELS. The latter value means that during the first (and possibly the only) attempt, the function uses the user-supplied labels as the initial approximation instead of generating random labels. For the second and further attempts, the function will use randomly generated labels in any case

centers The optional output array of the cluster centers

compactness The optional output parameter, which is computed as $\sum_i ||\texttt{samples}_i - \texttt{centers}_{\texttt{labels}_i}||^2$ after every attempt; the best (minimum) value is chosen and the corresponding labels are returned by the function. Basically, the user can use only the core of the function, set the number of attempts to 1, initialize labels each time using a custom algorithm (flags=CV_KMEAN_USE_INITIAL_I and, based on the output compactness or any other criteria, choose the best clustering.

The function cvKMeans2 implements a k-means algorithm that finds the centers of nclusters clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a cluster index for samples stored in the i-th row of the samples matrix.

```
#include "cxcore.h"
#include "highgui.h"

void main( int argc, char** argv )
{
    #define MAX_CLUSTERS 5
    CvScalar color_tab[MAX_CLUSTERS];
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG rng = cvRNG(0xffffffff);

    color_tab[0] = CV_RGB(255,0,0);
    color_tab[1] = CV_RGB(0,255,0);
    color_tab[2] = CV_RGB(100,100,255);
    color_tab[3] = CV_RGB(255,0,255);
    color_tab[4] = CV_RGB(255,255,0);

    cvNamedWindow( "clusters", 1 );

    for(;;)
    {
        int k, cluster_count = cvRandInt(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandInt(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );
```

```
        /* generate random sample from multigaussian distribution */
        for( k = 0; k < cluster_count; k++ )
        {
            CvPoint center;
            CvMat point_chunk;
            center.x = cvRandInt(&rng)%img->width;
            center.y = cvRandInt(&rng)%img->height;
            cvGetRows( points,
                       &point_chunk,
                       k*sample_count/cluster_count,
                       (k == (cluster_count - 1)) ?
                           sample_count :
                           (k+1)*sample_count/cluster_count );
            cvRandArr( &rng, &point_chunk, CV_RAND_NORMAL,
                       cvScalar(center.x,center.y,0,0),
                       cvScalar(img->width/6, img->height/6,0,0) );
        }

        /* shuffle samples */
        for( i = 0; i < sample_count/2; i++ )
        {
            CvPoint2D32f* pt1 =
                (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)%sample_count;
            CvPoint2D32f* pt2 =
                (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)%sample_count;
            CvPoint2D32f temp;
            CV_SWAP( *pt1, *pt2, temp );
        }

        cvKMeans2( points, cluster_count, clusters,
                   cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

        cvZero( img );

        for( i = 0; i < sample_count; i++ )
        {
            CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
            int cluster_idx = clusters->data.i[i];
            cvCircle( img,
                      cvPointFrom32f(pt),
                      2,
                      color_tab[cluster_idx],
                      CV_FILLED );
        }
```

```
        cvReleaseMat( &points );
        cvReleaseMat( &clusters );

        cvShowImage( "clusters", img );

        int key = cvWaitKey(0);
        if( key == 27 )
            break;
    }
}
```

---

cvSeqPartition

Splits a sequence into equivalency classes.

```
int cvSeqPartition(
            const CvSeq* seq,
            CvMemStorage* storage,
            CvSeq** labels,
            CvCmpFunc is_equal,
            void* userdata );
```

seq The sequence to partition

storage The storage block to store the sequence of equivalency classes. If it is NULL, the function uses `seq->storage` for output labels

labels Ouput parameter. Double pointer to the sequence of 0-based labels of input sequence elements

is_equal The relation function that should return non-zero if the two particular sequence elements are from the same class, and zero otherwise. The partitioning algorithm uses transitive closure of the relation function as an equivalency critria

userdata Pointer that is transparently passed to the `is_equal` function

```
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
```

The function `cvSeqPartition` implements a quadratic algorithm for splitting a set into one or more equivalancy classes. The function returns the number of equivalency classes.

```
#include "cxcore.h"
#include "highgui.h"
#include <stdio.h>

CvSeq* point_seq = 0;
IplImage* canvas = 0;
CvScalar* colors = 0;
int pos = 10;

int is_equal( const void* _a, const void* _b, void* userdata )
{
    CvPoint a = *(const CvPoint*)_a;
    CvPoint b = *(const CvPoint*)_b;
    double threshold = *(double*)userdata;
    return (double)((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y)) <=
        threshold;
}

void on_track( int pos )
{
    CvSeq* labels = 0;
    double threshold = pos*pos;
    int i, class_count = cvSeqPartition( point_seq,
                                         0,
                                         &labels,
                                         is_equal,
                                         &threshold );
    printf("%4d classes\n", class_count );
    cvZero( canvas );

    for( i = 0; i < labels->total; i++ )
    {
        CvPoint pt = *(CvPoint*)cvGetSeqElem( point_seq, i );
        CvScalar color = colors[*(int*)cvGetSeqElem( labels, i )];
        cvCircle( canvas, pt, 1, color, -1 );
    }

    cvShowImage( "points", canvas );
}

int main( int argc, char** argv )
{
    CvMemStorage* storage = cvCreateMemStorage(0);
    point_seq = cvCreateSeq( CV_32SC2,
```

```
                                sizeof(CvSeq),
                                sizeof(CvPoint),
                                storage );
    CvRNG rng = cvRNG(0xffffffff);

    int width = 500, height = 500;
    int i, count = 1000;
    canvas = cvCreateImage( cvSize(width,height), 8, 3 );

    colors = (CvScalar*)cvAlloc( count*sizeof(colors[0]) );
    for( i = 0; i < count; i++ )
    {
        CvPoint pt;
        int icolor;
        pt.x = cvRandInt( &rng ) % width;
        pt.y = cvRandInt( &rng ) % height;
        cvSeqPush( point_seq, &pt );
        icolor = cvRandInt( &rng ) | 0x00404040;
        colors[i] = CV_RGB(icolor & 255,
                           (icolor >> 8)&255,
                           (icolor >> 16)&255);
    }

    cvNamedWindow( "points", 1 );
    cvCreateTrackbar( "threshold", "points", &pos, 50, on_track );
    on_track(pos);
    cvWaitKey(0);
    return 0;
}
```

1.7 Utility and System Functions and Macros

Error Handling

Error handling in OpenCV is similar to IPL (Image Processing Library). In the case of an error, functions do not return the error code. Instead, they raise an error using `CV_ERROR` macro that calls cvError that, in its turn, sets the error status with cvSetErrStatus and calls a standard or user-defined error handler (that can display a message box, write to log, etc., see cvRedirectError). There is a global variable, one per each program thread, that contains current error status (an integer value). The status can be retrieved with the cvGetErrStatus function.

There are three modes of error handling (see cvSetErrMode and cvGetErrMode):

- **Leaf**. The program is terminated after the error handler is called. This is the default value.

It is useful for debugging, as the error is signalled immediately after it occurs. However, for production systems, other two methods may be preferable as they provide more control.

- **Parent**. The program is not terminated, but the error handler is called. The stack is unwound (it is done w/o using a C++ exception mechanism). The user may check error code after calling the `CxCore` function with cvGetErrStatus and react.
- **Silent**. Similar to `Parent` mode, but no error handler is called.

Actually, the semantics of the `Leaf` and `Parent` modes are implemented by error handlers and the above description is true for them. cvGuiBoxReport behaves slightly differently, and some custom error handlers may implement quite different semantics.

Macros for raising an error, checking for errors, etc.

```
/* special macros for enclosing processing statements within a function and separating
   them from prologue (resource initialization) and epilogue (guaranteed resource release)
#define __BEGIN__       {
#define __END__         goto exit; exit: ; }
/* proceeds to "resource release" stage */
#define EXIT            goto exit

/* Declares locally the function name for CV_ERROR() use */
#define CV_FUNCNAME( Name )  \
    static char cvFuncName[] = Name

/* Raises an error within the current context */
#define CV_ERROR( Code, Msg )                                           \
{                                                                       \
     cvError( (Code), cvFuncName, Msg, __FILE__, __LINE__ );            \
     EXIT;                                                              \
}

/* Checks status after calling CXCORE function */
#define CV_CHECK()                                                      \
{                                                                       \
    if( cvGetErrStatus() < 0 )                                      \
        CV_ERROR( CV_StsBackTrace, "Inner function failed." );      \
}

/* Provies shorthand for CXCORE function call and CV_CHECK() */
#define CV_CALL( Statement )                                            \
{                                                                       \
    Statement;                                                          \
    CV_CHECK();                                                         \
```

```
}

/* Checks some condition in both debug and release configurations */
#define CV_ASSERT( Condition )                                          \
{                                                                       \
    if( !(Condition) )                                                  \
        CV_ERROR( CV_StsInternal, "Assertion: " #Condition " failed" ); \
}

/* these macros are similar to their CV_... counterparts, but they
   do not need exit label nor cvFuncName to be defined */
#define OPENCV_ERROR(status,func_name,err_msg) ...
#define OPENCV_ERRCHK(func_name,err_msg) ...
#define OPENCV_ASSERT(condition,func_name,err_msg) ...
#define OPENCV_CALL(statement) ...
```

Instead of a discussion, below is a documented example of a typical CXCORE function and an example of the function use.

Example: Use of Error Handling Macros

```
#include "cxcore.h"
#include <stdio.h>

void cvResizeDCT( CvMat* input_array, CvMat* output_array )
{
    CvMat* temp_array = 0; // declare pointer that should be released anyway.

    CV_FUNCNAME( "cvResizeDCT" ); // declare cvFuncName

    __BEGIN__; // start processing. There may be some declarations just after
              // this macro, but they could not be accessed from the epilogue.

    if( !CV_IS_MAT(input_array) || !CV_IS_MAT(output_array) )
        // use CV_ERROR() to raise an error
        CV_ERROR( CV_StsBadArg,
        "input_array or output_array are not valid matrices" );

    // some restrictions that are going to be removed later, may be checked
    // with CV_ASSERT()
    CV_ASSERT( input_array->rows == 1 && output_array->rows == 1 );

    // use CV_CALL for safe function call
```

```
    CV_CALL( temp_array = cvCreateMat( input_array->rows,
                                       MAX(input_array->cols,
                                       output_array->cols),
                                       input_array->type ));

    if( output_array->cols > input_array->cols )
        CV_CALL( cvZero( temp_array ));

    temp_array->cols = input_array->cols;
    CV_CALL( cvDCT( input_array, temp_array, CV_DXT_FORWARD ));
    temp_array->cols = output_array->cols;
    CV_CALL( cvDCT( temp_array, output_array, CV_DXT_INVERSE ));
    CV_CALL( cvScale( output_array,
                      output_array,
                      1./sqrt((double)input_array->cols*output_array->cols), 0 ));

    __END__; // finish processing. Epilogue follows after the macro.

    // release temp_array. If temp_array has not been allocated
    // before an error occured, cvReleaseMat
    // takes care of it and does nothing in this case.
    cvReleaseMat( &temp_array );
}

int main( int argc, char** argv )
{
    CvMat* src = cvCreateMat( 1, 512, CV_32F );
#if 1 /* no errors */
    CvMat* dst = cvCreateMat( 1, 256, CV_32F );
#else
    CvMat* dst = 0; /* test error processing mechanism */
#endif
    cvSet( src, cvRealScalar(1.), 0 );
#if 0 /* change 0 to 1 to suppress error handler invocation */
    cvSetErrMode( CV_ErrModeSilent );
#endif
    cvResizeDCT( src, dst ); // if some error occurs, the message
                             // box will popup, or a message will be
                             // written to log, or some user-defined
                             // processing will be done
    if( cvGetErrStatus() < 0 )
        printf("Some error occured" );
    else
        printf("Everything is OK" );
    return 0;
```

```
}
```

cvGetErrStatus

Returns the current error status.

```
int cvGetErrStatus( void );
```

The function returns the current error status - the value set with the last cvSetErrStatus call. Note that in `Leaf` mode, the program terminates immediately after an error occurs, so to always gain control after the function call, one should call cvSetErrMode and set the `Parent` or `Silent` error mode.

cvSetErrStatus

Sets the error status.

```
void cvSetErrStatus( int status );
```

`status` The error status

The function sets the error status to the specified value. Mostly, the function is used to reset the error status (set to it `CV_StsOk`) to recover after an error. In other cases it is more natural to call cvError or `CV_ERROR`.

cvGetErrMode

Returns the current error mode.

```
int cvGetErrMode(void);
```

The function returns the current error mode - the value set with the last cvSetErrMode call.

cvSetErrMode

Sets the error mode.

```
#define CV_ErrModeLeaf    0
#define CV_ErrModeParent  1
#define CV_ErrModeSilent  2
```

```
int cvSetErrMode( int mode );
```

mode The error mode

The function sets the specified error mode. For descriptions of different error modes, see the beginning of the error section.

cvError

Raises an error.

```
int cvError(
            int status,
            const char* func_name,
            const char* err_msg,
            const char* filename,
            int line );
```

status The error status

func_name Name of the function where the error occured

err_msg Additional information/diagnostics about the error

filename Name of the file where the error occured

line Line number, where the error occured

The function sets the error status to the specified value (via cvSetErrStatus) and, if the error mode is not `Silent`, calls the error handler.

cvErrorStr

Returns textual description of an error status code.

```
const char* cvErrorStr( int status );
```

status The error status

The function returns the textual description for the specified error status code. In the case of unknown status, the function returns a NULL pointer.

cvRedirectError

Sets a new error handler.

```
CvErrorCallback cvRedirectError(
            CvErrorCallback error_handler,
            void* userdata=NULL,
            void** prevUserdata=NULL );
```

error_handler The new error_handler

userdata Arbitrary pointer that is transparently passed to the error handler

prevUserdata Pointer to the previously assigned user data pointer

```
typedef int (CV_CDECL *CvErrorCallback)( int status, const char* func_name,
                    const char* err_msg, const char* file_name, int line );
```

The function sets a new error handler that can be one of the standard handlers or a custom handler that has a specific interface. The handler takes the same parameters as the cvError function. If the handler returns a non-zero value, the program is terminated; otherwise, it continues. The error handler may check the current error mode with cvGetErrMode to make a decision.

cvNulDevReport cvStdErrReport cvGuiBoxReport

Provide standard error handling.

```
int cvNulDevReport( int status, const char* func_name, const char*
err_msg, const char* file_name, int line, void* userdata );

          int cvStdErrReport( int status, const char* func_name, const
char* err_msg, const char* file_name, int line, void* userdata );

          int cvGuiBoxReport( int status, const char* func_name, const
char* err_msg, const char* file_name, int line, void* userdata );
```

status The error status

func_name Name of the function where the error occured

err_msg Additional information/diagnostics about the error

filename Name of the file where the error occured

line Line number, where the error occured

userdata Pointer to the user data. Ignored by the standard handlers

The functions `cvNullDevReport`, `cvStdErrReport`, and `cvGuiBoxReport` provide standard error handling. `cvGuiBoxReport` is the default error handler on Win32 systems, `cvStdErrReport` is the default on other systems. `cvGuiBoxReport` pops up a message box with the error description and suggest a few options. Below is an example message box that may be recieved with the sample code above, if one introduces an error as described in the sample.

Error Message Box

If the error handler is set to `cvStdErrReport`, the above message will be printed to standard error output and the program will be terminated or continued, depending on the current error mode.

Error Message printed to Standard Error Output (in `Leaf` mode)

```
OpenCV ERROR: Bad argument (input_array or output_array are not valid matrices)
        in function cvResizeDCT, D:\User\VP\Projects\avl\_proba\a.cpp(75)
Terminating the application...
```

cvAlloc

Allocates a memory buffer.

```
void* cvAlloc( size_t size );
```

size Buffer size in bytes

The function allocates `size` bytes and returns a pointer to the allocated buffer. In the case of an error the function reports an error and returns a NULL pointer. By default, `cvAlloc` calls `icvAlloc` which itself calls `malloc`. However it is possible to assign user-defined memory allocation/deallocation functions using the cvSetMemoryManager function.

cvFree

Deallocates a memory buffer.

```
void cvFree( void** ptr );
```

ptr Double pointer to released buffer

The function deallocates a memory buffer allocated by cvAlloc. It clears the pointer to buffer upon exit, which is why the double pointer is used. If the `*buffer` is already NULL, the function does nothing.

cvGetTickCount

Returns the number of ticks.

```
int64 cvGetTickCount( void );
```

The function returns number of the ticks starting from some platform-dependent event (number of CPU ticks from the startup, number of milliseconds from 1970th year, etc.). The function is useful for accurate measurement of a function/user-code execution time. To convert the number of ticks to time units, use cvGetTickFrequency.

cvGetTickFrequency

Returns the number of ticks per microsecond.

```
double cvGetTickFrequency( void );
```

The function returns the number of ticks per microsecond. Thus, the quotient of cvGetTickCount and cvGetTickFrequency will give the number of microseconds starting from the platform-dependent event.

cvRegisterModule

Registers another module.

```
typedef struct CvPluginFuncInfo
{
    void** func_addr;
    void* default_func_addr;
    const char* func_names;
    int search_modules;
    int loaded_from;
}
CvPluginFuncInfo;

typedef struct CvModuleInfo
{
    struct CvModuleInfo* next;
    const char* name;
    const char* version;
    CvPluginFuncInfo* func_tab;
}
CvModuleInfo;
```

```
int cvRegisterModule( const CvModuleInfo* moduleInfo );
```

moduleInfo Information about the module

The function adds a module to the list of registered modules. After the module is registered, information about it can be retrieved using the cvGetModuleInfo function. Also, the registered

module makes full use of optimized plugins (IPP, MKL, ...), supported by CXCORE. CXCORE itself, CV (computer vision), CVAUX (auxilary computer vision), and HIGHGUI (visualization and image/video acquisition) are examples of modules. Registration is usually done when the shared library is loaded. See `cxcore/src/cxswitcher.cpp` and `cv/src/cvswitcher.cpp` for details about how registration is done and look at `cxcore/src/cxswitcher.cpp`, `cxcore/src/_cxipp.h` on how IPP and MKL are connected to the modules.

---

cvGetModuleInfo

Retrieves information about registered module(s) and plugins.

```
void cvGetModuleInfo(
           const char* moduleName,
           const char** version,
           const char** loadedAddonPlugins);
```

moduleName Name of the module of interest, or NULL, which means all the modules

version The output parameter. Information about the module(s), including version

loadedAddonPlugins The list of names and versions of the optimized plugins that CXCORE was able to find and load

The function returns information about one or all of the registered modules. The returned information is stored inside the libraries, so the user should not deallocate or modify the returned text strings.

---

cvUseOptimized

Switches between optimized/non-optimized modes.

```
int cvUseOptimized( int onoff );
```

onoff Use optimized ($\neq 0$) or not ($= 0$)

The function switches between the mode, where only pure C implementations from cxcore, OpenCV, etc. are used, and the mode, where IPP and MKL functions are used if available. When

`cvUseOptimized(0)` is called, all the optimized libraries are unloaded. The function may be useful for debugging, IPP and MKL upgrading on the fly, online speed comparisons, etc. It returns the number of optimized functions loaded. Note that by default, the optimized plugins are loaded, so it is not necessary to call `cvUseOptimized(1)` in the beginning of the program (actually, it will only increase the startup time).

---

cvSetMemoryManager

Accesses custom/default memory managing functions.

```
typedef void* (CV_CDECL *CvAllocFunc)(size_t size, void* userdata);
typedef int (CV_CDECL *CvFreeFunc)(void* pptr, void* userdata);
```

```
void cvSetMemoryManager(
           CvAllocFunc allocFunc=NULL,
           CvFreeFunc freeFunc=NULL,
           void* userdata=NULL );
```

allocFunc Allocation function; the interface is similar to `malloc`, except that `userdata` may be used to determine the context

freeFunc Deallocation function; the interface is similar to `free`

userdata User data that is transparently passed to the custom functions

The function sets user-defined memory managment functions (substitutes for `malloc` and `free`) that will be called by `cvAlloc, cvFree` and higher-level functions (e.g., `cvCreateImage`). Note that the function should be called when there is data allocated using `cvAlloc`. Also, to avoid infinite recursive calls, it is not allowed to call `cvAlloc` and cvFree from the custom allocation/deallocation functions.

If the `alloc_func` and `free_func` pointers are `NULL`, the default memory managing functions are restored.

---

cvSetIPLAllocators

Switches to IPL functions for image allocation/deallocation.

```
typedef IplImage* (CV_STDCALL* Cv_iplCreateImageHeader)
                            (int,int,int,char*,char*,int,int,int,int,int,
                            IplROI*,IplImage*,void*,IplTileInfo*);
```

```
typedef void (CV_STDCALL* Cv_iplAllocateImageData)(IplImage*,int,int);
typedef void (CV_STDCALL* Cv_iplDeallocate)(IplImage*,int);
typedef IplROI* (CV_STDCALL* Cv_iplCreateROI)(int,int,int,int,int);
typedef IplImage* (CV_STDCALL* Cv_iplCloneImage)(const IplImage*);

#define CV_TURN_ON_IPL_COMPATIBILITY()                                      \
    cvSetIPLAllocators( iplCreateImageHeader, iplAllocateImage,         \
                        iplDeallocate, iplCreateROI, iplCloneImage )
```

```
void cvSetIPLAllocators(
           Cv_iplCreateImageHeader create_header,
           Cv_iplAllocateImageData allocate_data,
           Cv_iplDeallocate deallocate,
           Cv_iplCreateROI create_roi,
           Cv_iplCloneImage clone_image );
```

create_header Pointer to iplCreateImageHeader

allocate_data Pointer to iplAllocateImage

deallocate Pointer to iplDeallocate

create_roi Pointer to iplCreateROI

clone_image Pointer to iplCloneImage

The function causes CXCORE to use IPL functions for image allocation/deallocation operations. For convenience, there is the wrapping macro `CV_TURN_ON_IPL_COMPATIBILITY`. The function is useful for applications where IPL and CXCORE/OpenCV are used together and still there are calls to `iplCreateImageHeader`, etc. The function is not necessary if IPL is called only for data processing and all the allocation/deallocation is done by CXCORE, or if all the allocation/deallocation is done by IPL and some of OpenCV functions are used to process the data.

Chapter 2

cv. Image Processing and Computer Vision

2.1 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as cv's), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y). It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc.

IplConvKernel

An IplConvKernel is a rectangular convolution kernel, created by function CreateStructuringElementEx .

cvCopyMakeBorder

Copies an image and makes a border around it.

```
void cvCopyMakeBorder(
           const CvArr* src,
           CvArr* dst,
           CvPoint offset,
           int bordertype,
           CvScalar value=cvScalarAll(0) );
```

src The source image

dst The destination image

offset Coordinates of the top-left corner (or bottom-left in the case of images with bottom-left origin) of the destination image rectangle where the source image (or its ROI) is copied. Size of the rectanlge matches the source image size/ROI size

bordertype Type of the border to create around the copied source image rectangle; types include:

IPL_BORDER_CONSTANT border is filled with the fixed value, passed as last parameter of the function.

IPL_BORDER_REPLICATE the pixels from the top and bottom rows, the left-most and right-most columns are replicated to fill the border.

(The other two border types from IPL, `IPL_BORDER_REFLECT` and `IPL_BORDER_WRAP`, are currently unsupported)

value Value of the border pixels if `bordertype` is `IPL_BORDER_CONSTANT`

The function copies the source 2D array into the interior of the destination array and makes a border of the specified type around the copied area. The function is useful when one needs to emulate border type that is different from the one embedded into a specific algorithm implementation. For example, morphological functions, as well as most of other filtering functions in OpenCV, internally use replication border type, while the user may need a zero border or a border, filled with 1's or 255's.

cvCreateStructuringElementEx

Creates a structuring element.

```
IplConvKernel* cvCreateStructuringElementEx(
            int cols,
            int rows,
            int anchorX,
            int anchorY,
            int shape,
            int* values=NULL );
```

cols Number of columns in the structuring element

rows Number of rows in the structuring element

anchorX Relative horizontal offset of the anchor point

anchorY Relative vertical offset of the anchor point

shape Shape of the structuring element; may have the following values:

CV_SHAPE_RECT a rectangular element

CV_SHAPE_CROSS a cross-shaped element

CV_SHAPE_ELLIPSE an elliptic element

CV_SHAPE_CUSTOM a user-defined element. In this case the parameter `values` specifies the mask, that is, which neighbors of the pixel must be considered

values Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non-zero values indicate points that belong to the element. If the pointer is `NULL`, then all values are considered non-zero, that is, the element is of a rectangular shape. This parameter is considered only if the shape is `CV_SHAPE_CUSTOM`

The function CreateStructuringElementEx allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the morphological operations.

cvDilate

Dilates an image by using a specific structuring element.

```
void cvDilate(
           const CvArr* src,
           CvArr* dst,
           IplConvKernel* element=NULL,
           int iterations=1 );
```

src Source image

dst Destination image

element Structuring element used for dilation. If it is NULL, a 3×3 rectangular structuring element is used

iterations Number of times dilation is applied

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\max_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (iterations) times. For color images, each channel is processed independently.

cvErode

Erodes an image by using a specific structuring element.

```
void cvErode(
           const CvArr* src,
           CvArr* dst,
           IplConvKernel* element=NULL,
           int iterations=1);
```

src Source image

dst Destination image

element Structuring element used for erosion. If it is NULL, a 3×3 rectangular structuring element is used

iterations Number of times erosion is applied

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\min_{(x',y') \, in \, element} src(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (iterations) times. For color images, each channel is processed independently.

---

cvFilter2D

Convolves an image with the kernel.

```
void cvFilter2D(
           const CvArr* src,
           CvArr* dst,
           const CvMat* kernel,
           CvPoint anchor=cvPoint(-1,-1));
```

src The source image

dst The destination image

kernel Convolution kernel, a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cvSplit and process them individually

anchor The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor shoud lie within the kernel. The special default value (-1,-1) means that it is at the kernel center

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values from the nearest pixels that are inside the image.

cvLaplace

Calculates the Laplacian of an image.

```
void cvLaplace(
           const CvArr* src,
           CvArr* dst,
           int apertureSize=3);
```

src Source image

dst Destination image

apertureSize Aperture size (it has the same meaning as cvSobel)

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst}(x,y) = \frac{d^2\texttt{src}}{dx^2} + \frac{d^2\texttt{src}}{dy^2}$$

Setting `apertureSize` = 1 gives the fastest variant that is equal to convolving the image with the following kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Similar to the cvSobel function, no scaling is done and the same combinations of input and output formats are supported.

cvMorphologyEx

Performs advanced morphological transformations.

```
void cvMorphologyEx(
           const CvArr* src,
           CvArr* dst,
           CvArr* temp,
```

```
            IplConvKernel* element,
            int operation,
            int iterations=1 );
```

src Source image

dst Destination image

temp Temporary image, required in some cases

element Structuring element

operation Type of morphological operation, one of the following:

CV_MOP_OPEN opening

CV_MOP_CLOSE closing

CV_MOP_GRADIENT morphological gradient

CV_MOP_TOPHAT "top hat"

CV_MOP_BLACKHAT "black hat"

iterations Number of times erosion and dilation are applied

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$dst = open(src, element) = dilate(erode(src, element), element)$$

Closing:

$$dst = close(src, element) = erode(dilate(src, element), element)$$

Morphological gradient:

$$dst = morph\_grad(src, element) = dilate(src, element) - erode(src, element)$$

"Top hat":

$$dst = tophat(src, element) = src - open(src, element)$$

"Black hat":

$$dst = blackhat(src, element) = close(src, element) - src$$

The temporary image `temp` is required for a morphological gradient and, in the case of in-place operation, for ”top hat” and ”black hat”.

---

cvPyrDown

Downsamples an image.

```
void cvPyrDown(
           const CvArr* src,
           CvArr* dst,
           int filter=CV_GAUSSIAN_5x5 );
```

src The source image

dst The destination image, should have a half as large width and height than the source

filter Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported

The function performs the downsampling step of the Gaussian pyramid decomposition. First it convolves the source image with the specified filter and then downsamples the image by rejecting even rows and columns.

---

cvReleaseStructuringElement

Deletes a structuring element.

```
void cvReleaseStructuringElement( IplConvKernel** element );
```

element Pointer to the deleted structuring element

The function releases the structure `IplConvKernel` that is no longer needed. If `*element` is `NULL`, the function has no effect.

cvSmooth

Smooths the image in one of several ways.

```
void cvSmooth(
           const CvArr* src,
           CvArr* dst,
           int smoothtype=CV_GAUSSIAN,
           int param1=3,
           int param2=0,
           double param3=0,
           double param4=0);
```

src The source image

dst The destination image

smoothtype Type of the smoothing:

- **CV_BLUR_NO_SCALE** linear convolution with `param1` × `param2` box kernel (all 1's). If you want to smooth different pixels with different-size box kernels, you can use the integral image that is computed using cvIntegral
- **CV_BLUR** linear convolution with `param1` × `param2` box kernel (all 1's) with subsequent scaling by $1/(\texttt{param1} \cdot \texttt{param2})$
- **CV_GAUSSIAN** linear convolution with a `param1` × `param2` Gaussian kernel
- **CV_MEDIAN** median filter with a `param1` × `param1` square aperture
- **CV_BILATERAL** bilateral filter with a `param1`×`param1` square aperture, color sigma=`param3` and spatial sigma=`param4`. If `param1=0`, the aperture square side is set to `cvRound(param4*1.5)*` Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1 The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)

param2 The second parameter of the smoothing operation, the aperture height. Ignored by `CV_MEDIAN` and `CV_BILATERAL` methods. In the case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1`. Otherwise it must be a positive odd number.

param3 In the case of a Gaussian parameter this parameter may specify Gaussian σ (standard deviation). If it is zero, it is calculated from the kernel size:

$$\sigma = 0.3(n/2 - 1) + 0.8 \quad \text{where} \quad n = \begin{array}{l} \texttt{param1} \text{ for horizontal kernel} \\ \texttt{param2} \text{ for vertical kernel} \end{array}$$

Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function smooths an image using one of several methods. Every of the methods has some features and restrictions listed below

Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to cvSobel and cvLaplace) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

---

cvSobel

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator.

```
void cvSobel(
           const CvArr* src,
           CvArr* dst,
           int xorder,
           int yorder,
           int apertureSize=3 );
```

src Source image of type CvArr*

dst Destination image

xorder Order of the derivative x

yorder Order of the derivative y

apertureSize Size of the extended Sobel kernel, must be 1, 3, 5 or 7

In all cases except 1, an `apertureSize` $\times$ `apertureSize` separable kernel will be used to calculate the derivative. For `apertureSize` $= 1$ a 3×1 or 1×3 a kernel is used (Gaussian smoothing is not done). There is also the special value `CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\texttt{dst}(x,y) = \frac{d^{xorder+yorder}\,\texttt{src}}{dx^{xorder} \cdot dy^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `apertureSize` = 3) or (`xorder` = 0, `yorder` = 1, `apertureSize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and the second one corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

or a kernel of:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix}$$

depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger numbers (in absolute values) than the source image does. To avoid overflow, the function requires a 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using the cvConvertScale or the cvConvertScaleAbs function. Besides 8-bit images the function can process 32-bit floating-point images. Both the source and the destination must be single-channel images of equal size or equal ROI size.

2.2 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x, y) = \texttt{src}(f_x(x, y), f_y(x, y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic cvRemap and to the simplest and the fastest cvResize, need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x, y) one of $f_x(x, y)$ or $f_y(x, y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method `BORDER_TRANSPARENT`, which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x, y)$ and $f_y(x, y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x, y), f_y(x, y))$ and then the value of the polynomial at $(f_x(x, y), f_y(x, y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see cvResize.

---

cvGetRotationMatrix2D

Calculates the affine matrix of 2d rotation.

```
CvMat* cv2DRotationMatrix(
           CvPoint2D32f center,
           double angle,
           double scale,
           CvMat* mapMatrix );
```

center Center of the rotation in the source image

angle The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)

scale Isotropic scale factor

mapMatrix Pointer to the destination 2×3 matrix

The function `cv2DRotationMatrix` calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} - (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot cos(\texttt{angle}), \beta = \texttt{scale} \cdot sin(\texttt{angle})$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

---

cvGetAffineTransform

Calculates the affine transform from 3 corresponding points.

```
CvMat* cvGetAffineTransform(
           const CvPoint2D32f* src,
           const CvPoint2D32f* dst,
           CvMat* mapMatrix );
```

src Coordinates of 3 triangle vertices in the source image

dst Coordinates of the 3 corresponding triangle vertices in the destination image

mapMatrix Pointer to the destination 2×3 matrix

The function cvGetAffineTransform calculates the matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

---

cvGetPerspectiveTransform

Calculates the perspective transform from 4 corresponding points.

```
CvMat* cvGetPerspectiveTransform(
           const CvPoint2D32f* src,
           const CvPoint2D32f* dst,
           CvMat* mapMatrix );
```

src Coordinates of 4 quadrangle vertices in the source image

dst Coordinates of the 4 corresponding quadrangle vertices in the destination image

mapMatrix Pointer to the destination 3×3 matrix

The function `cvGetPerspectiveTransform` calculates a matrix of perspective transforms such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

cvGetQuadrangleSubPix

Retrieves the pixel quadrangle from an image with sub-pixel accuracy.

```
void cvGetQuadrangleSubPix(
           const CvArr* src,
           CvArr* dst,
           const CvMat* mapMatrix );
```

src Source image

dst Extracted quadrangle

mapMatrix The transformation 2×3 matrix $[A|b]$ (see the discussion)

The function `cvGetQuadrangleSubPix` extracts pixels from `src` at sub-pixel accuracy and stores them to `dst` as follows:

$$dst(x, y) = src(A_{11}x' + A_{12}y' + b_1, A_{21}x' + A_{22}y' + b_2)$$

where

$$x' = x - \frac{(width(dst) - 1)}{2}, y' = y - \frac{(height(dst) - 1)}{2}$$

and

$$\texttt{mapMatrix} = \begin{bmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{bmatrix}$$

The values of pixels at non-integer coordinates are retrieved using bilinear interpolation. When the function needs pixels outside of the image, it uses replication border mode to reconstruct the values. Every channel of multiple-channel images is processed independently.

cvGetRectSubPix

Retrieves the pixel rectangle from an image with sub-pixel accuracy.

```
void cvGetRectSubPix(
           const CvArr* src,
           CvArr* dst,
           CvPoint2D32f center );
```

src Source image

dst Extracted rectangle

center Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image

The function `cvGetRectSubPix` extracts pixels from `src`:

$$dst(x, y) = src(x + \texttt{center.x} - (width(\texttt{dst}) - 1) * 0.5, y + \texttt{center.y} - (height(\texttt{dst}) - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

---

cvLogPolar

Remaps an image to log-polar space.

```
void cvLogPolar(
           const CvArr* src,
           CvArr* dst,
           CvPoint2D32f center,
           double M,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
```

src Source image

dst Destination image

center The transformation center; where the output precision is maximal

M Magnitude scale parameter. See below

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero

CV_WARP_INVERSE_MAP See below

The function `cvLogPolar` transforms the source image using the following transformation:
Forward transformation (`CV_WARP_INVERSE_MAP` is not set):

$$dst(\phi, \rho) = src(x, y)$$

Inverse transformation (`CV_WARP_INVERSE_MAP` is set):

$$dst(x, y) = src(\phi, \rho)$$

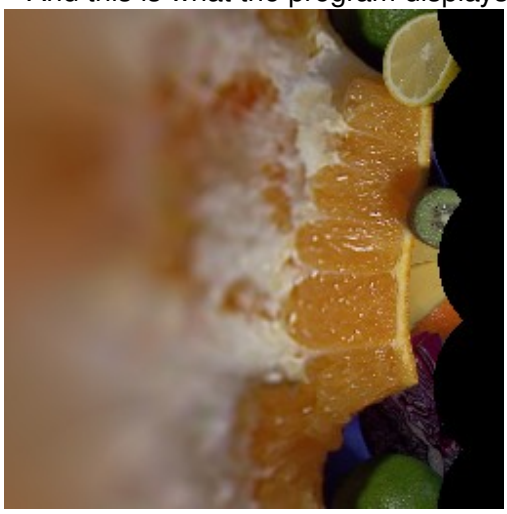
where

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

```
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv)
{
    IplImage* src;

    if( argc == 2 && (src=cvLoadImage(argv[1],1) != 0 )
    {
        IplImage* dst = cvCreateImage( cvSize(256,256), 8, 3 );
        IplImage* src2 = cvCreateImage( cvGetSize(src), 8, 3 );
        cvLogPolar( src, dst, cvPoint2D32f(src->width/2,src->height/2), 40,
        CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
        cvLogPolar( dst, src2, cvPoint2D32f(src->width/2,src->height/2), 40,
        CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS+CV_WARP_INVERSE_MAP );
        cvNamedWindow( "log-polar", 1 );
        cvShowImage( "log-polar", dst );
        cvNamedWindow( "inverse log-polar", 1 );
        cvShowImage( "inverse log-polar", src2 );
        cvWaitKey();
    }
    return 0;
}
```

And this is what the program displays when `opencv/samples/c/fruits.jpg` is passed to it



---

cvRemap

Applies a generic geometrical transformation to the image.

```
void cvRemap(
           const CvArr* src,
           CvArr* dst,
           const CvArr* mapx,
           const CvArr* mapy,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
           CvScalar fillval=cvScalarAll(0) );
```

src Source image

dst Destination image

mapx The map of x-coordinates (CV_32FC1 image)

mapy The map of y-coordinates (CV_32FC1 image)

flags A combination of interpolation method and the following optional flag(s):

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`

fillval A value used to fill outliers

The function cvRemap transforms the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(\texttt{mapx}(x,y), \texttt{mapy}(x,y))$$

Similar to other geometrical transformations, some interpolation method (specified by user) is used to extract pixels with non-integer coordinates. Note that the function can not operate in-place.

---

cvResize

Resizes an image.

```
void cvResize(
           const CvArr* src,
           CvArr* dst,
           int interpolation=CV_INTER_LINEAR );
```

src Source image

dst Destination image

interpolation Interpolation method:

- **CV_INTER_NN** nearest-neigbor interpolation
- **CV_INTER_LINEAR** bilinear interpolation (used by default)
- **CV_INTER_AREA** resampling using pixel area relation. It is the preferred method for image decimation that gives moire-free results. In terms of zooming it is similar to the CV_INTER_NN method
- **CV_INTER_CUBIC** bicubic interpolation

The function cvResize resizes an image src so that it fits exactly into dst. If ROI is set, the function considers the ROI as supported.

---

cvWarpAffine

Applies an affine transformation to an image.

```
void cvWarpAffine(
           const CvArr* src,
           CvArr* dst,
           const CvMat* mapMatrix,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
           CvScalar fillval=cvScalarAll(0) );
```

src Source image

dst Destination image

mapMatrix 2×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`

CV_WARP_INVERSE_MAP indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpAffine` transforms the source image using the specified matrix:

$$dst(x', y') = src(x, y)$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV\_WARP\_INVERSE\_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

The function is similar to cvGetQuadrangleSubPix but they are not exactly the same. cvWarpAffine requires input and output image have the same data type, has larger overhead (so it is not quite suitable for small images) and can leave part of destination image unchanged. While cvGetQuadrangleSubPix may extract quadrangles from 8-bit images into floating-point buffer, has smaller

overhead and always changes the whole destination image content. Note that the function can not operate in-place.

To transform a sparse set of points, use the cvTransform function from cxcore.

---

cvWarpPerspective

Applies a perspective transformation to an image.

```
void cvWarpPerspective(
           const CvArr* src,
           CvArr* dst,
           const CvMat* mapMatrix,
           int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
           CvScalar fillval=cvScalarAll(0) );
```

src Source image

dst Destination image

mapMatrix 3×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

- **CV_WARP_FILL_OUTLIERS** fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`
- **CV_WARP_INVERSE_MAP** indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpPerspective` transforms the source image using the specified matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV\_WARP\_INVERSE\_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

Note that the function can not operate in-place. For a sparse set of points use the cvPerspectiveTransform function from CxCore.

2.3 Miscellaneous Image Transformations

---

cvAdaptiveThreshold

Applies an adaptive threshold to an array.

```
void cvAdaptiveThreshold(
           const CvArr* src,
           CvArr* dst,
           double maxValue,
           int adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C,
           int thresholdType=CV_THRESH_BINARY,
           int blockSize=3,
           double param1=5 );
```

src Source image

dst Destination image

maxValue Maximum value that is used with CV_THRESH_BINARY and CV_THRESH_BINARY_INV

adaptive_method Adaptive thresholding algorithm to use: CV_ADAPTIVE_THRESH_MEAN_C or CV_ADAPTIVE_THRESH_GAUSSIAN_C (see the discussion)

thresholdType Thresholding type; must be one of

CV_THRESH_BINARY xxx

CV_THRESH_BINARY_INV xxx

blockSize The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

param1 The method-dependent parameter. For the methods CV_ADAPTIVE_THRESH_MEAN_C and CV_ADAPTIVE_THRESH_GAUSSIAN_C it is a constant subtracted from the mean or weighted mean (see the discussion), though it may be negative

The function transforms a grayscale image to a binary image according to the formulas:

CV_THRESH_BINARY

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

For the method `CV_ADAPTIVE_THRESH_MEAN_C` it is the mean of a `blockSize` × `blockSize` pixel neighborhood, minus `param1`.

For the method `CV_ADAPTIVE_THRESH_GAUSSIAN_C` it is the weighted sum (gaussian) of a `blockSize` × `blockSize` pixel neighborhood, minus `param1`.

---

cvCvtColor

Converts an image from one color space to another.

```
void cvCvtColor(
           const CvArr* src,
           CvArr* dst,
           int code );
```

src The source 8-bit (8u), 16-bit (16u) or single-precision floating-point (32f) image

dst The destination image of the same data type as the source. The number of channels may be different

code Color conversion operation that can be specifed using `CV_` *src_color_space* `2` *dst_color_space* constants (see below)

The function converts the input image from one color space to another. The function ignores the `colorModel` and `channelSeq` fields of the `IplImage` header, so the source image color space should be specified correctly (including order of the channels in the case of RGB space. For example, BGR means 24-bit format with $B_0, G_0, R_0, B_1, G_1, R_1, ...$ layout whereas RGB means 24-format with $R_0, G_0, B_0, R_1, G_1, B_1, ...$ layout).

The conventional range for R,G,B channel values is:

- 0 to 255 for 8-bit images
- 0 to 65535 for 16-bit images and
- 0 to 1 for floating-point images.

Of course, in the case of linear transformations the range can be specific, but in order to get correct results in the case of non-linear transformations, the input image should be scaled.

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

  ```
  cvCvtColor(src ,bwsrc, CV_RGB2GRAY)
  ```

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ`, `CV_RGB2XYZ`, `CV_XYZ2BGR`, `CV_XYZ2RGB`):

  $$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

  $$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB $\leftrightarrow$ YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb`, `CV_RGB2YCrCb`, `CV_YCrCb2BGR`, `CV_YCrCb2RGB`)

  $$\begin{aligned} Y &\leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ Cr &\leftarrow (R - Y) \cdot 0.713 + delta \\ Cb &\leftarrow (B - Y) \cdot 0.564 + delta \\ R &\leftarrow Y + 1.403 \cdot (Cr - delta) \\ G &\leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta) \\ B &\leftarrow Y + 1.773 \cdot (Cb - delta) \end{aligned}$$

  where

  $$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

  Y, Cr and Cb cover the whole value range.

- RGB ↔ HSV (CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R,G,B)$$

$$S \leftarrow \begin{cases} \frac{V-min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V=R \\ 120+60(B-R)/S & \text{if } V=G \\ 240+60(R-G)/S & \text{if } V=B \end{cases}$$

if $H<0$ then $H \leftarrow H+360$

On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB ↔ HLS (CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R,G,B)$$

$$V_{min} \leftarrow min(R,G,B)$$

$$L \leftarrow \frac{V_{max}-V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max}-V_{min}}{V_{max}+V_{min}} & \text{if } L<0.5 \\ \frac{V_{max}-V_{min}}{2-(V_{max}+V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V_{max}=R \\ 120+60(B-R)/S & \text{if } V_{max}=G \\ 240+60(R-G)/S & \text{if } V_{max}=B \end{cases}$$

if $H<0$ then $H \leftarrow H+360$ On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2\text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \le 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t <= 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \le L \le 100$, $-127 \le a \le 127$, $-127 \le b \le 127$

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow L * 255/100, a \leftarrow a + 128, b \leftarrow b + 128$$

16-bit images currently not supported

32-bit images L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv`, `CV_RGB2Luv`, `CV_Luv2BGR`, `CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y <= 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow 255/100L, u \leftarrow 255/354(u + 134), v \leftarrow 255/256(v + 140)$$

16-bit images currently not supported

32-bit images L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Ford98 at the Charles Poynton site.

- Bayer $\rightarrow$ RGB (`CV_BayerBG2BGR`, `CV_BayerGB2BGR`, `CV_BayerRG2BGR`, `CV_BayerGR2BGR`, `CV_BayerBG2RGB`, `CV_BayerGB2RGB`, `CV_BayerRG2RGB`, `CV_BayerGR2RGB`) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{matrix} R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants `CV_Bayer` C_1C_2 `2BGR` and `CV_Bayer` C_1C_2 `2RGB` indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

---

cvDistTransform

Calculates the distance to the closest zero pixel for all non-zero pixels of the source image.

```
void cvDistTransform(
           const CvArr* src,
           CvArr* dst,
           int distance_type=CV_DIST_L2,
           int mask_size=3,
           const float* mask=NULL,
           CvArr* labels=NULL );
```

src 8-bit, single-channel (binary) source image

dst Output image with calculated distances (32-bit floating-point, single-channel)

distance_type Type of distance; can be `CV_DIST_L1, CV_DIST_L2, CV_DIST_C` or `CV_DIST_USER`

mask_size Size of the distance transform mask; can be 3 or 5. in the case of `CV_DIST_L1` or `CV_DIST_C` the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 yet it is faster

mask User-defined mask in the case of a user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in the case ofa 3×3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight's move cost) in the case of a 5×5 mask

labels The optional output 2d array of integer type labels, the same size as `src` and `dst`

The function calculates the approximated distance from every binary image pixel to the nearest zero pixel. For zero pixels the function sets the zero distance, for others it finds the shortest path consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the

distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (that is denoted as `a`), all the diagonal shifts must have the same cost (denoted `b`), and all knight's moves must have the same cost (denoted `c`). For `CV_DIST_C` and `CV_DIST_L1` types the distance is calculated precisely, whereas for `CV_DIST_L2` (Euclidian distance) the distance can be calculated only with some relative error (a 5×5 mask gives more accurate results), OpenCV uses the values suggested in [**?**]:

`CV_DIST_C`	(3×3)	a = 1, b = 1
`CV_DIST_L1`	(3×3)	a = 1, b = 2
`CV_DIST_L2`	(3×3)	a=0.955, b=1.3693
`CV_DIST_L2`	(5×5)	a=1, b=1.4, c=2.1969

And below are samples of the distance field (black (0) pixel is in the middle of white square) in the case of a user-defined distance:

User-defined 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1		1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

User-defined 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1		1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

Typically, for a fast, coarse distance estimation `CV_DIST_L2`, a 3×3 mask is used, and for a more accurate distance estimation `CV_DIST_L2`, a 5×5 mask is used.

When the output parameter `labels` is not `NULL`, for every non-zero pixel the function also finds the nearest connected component consisting of zero pixels. The connected components themselves are found as contours in the beginning of the function.

In this mode the processing time is still O(N), where N is the number of pixels. Thus, the function provides a very fast way to compute approximate Voronoi diagram for the binary image.

CvConnectedComp

```
typedef struct CvConnectedComp
```

```
{
    double area;    /* area of the segmented component */
    CvScalar value; /* average color of the connected component */
    CvRect rect;    /* ROI of the segmented component */
    CvSeq* contour; /* optional component boundary
                       (the contour might have child contours corresponding to the holes) */
} CvConnectedComp;
```

cvFloodFill

Fills a connected component with the given color.

```
void cvFloodFill(
           CvArr* image,
           CvPoint seed_point,
           CvScalar new_val,
           CvScalar lo_diff=cvScalarAll(0),
           CvScalar up_diff=cvScalarAll(0),
           CvConnectedComp* comp=NULL,
           int flags=4,
           CvArr* mask=NULL );
```

image Input 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the CV_FLOODFILL_MASK_ONLY flag is set (see below)

seed_point The starting point

new_val New value of the repainted domain pixels

lo_diff Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

up_diff Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

comp Pointer to the structure that the function fills with the information about the repainted domain

flags The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:

CV_FLOODFILL_FIXED_RANGE if set, the difference between the current pixel and seed pixel is considered, otherwise the difference between neighbor pixels is considered (the range is floating)

CV_FLOODFILL_MASK_ONLY if set, the function does not fill the image (`new_val` is ignored), but fills the mask (that must be non-NULL in this case)

mask Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller than `image`. If not NULL, the function uses and updates the mask, so the user takes responsibility of initializing the `mask` content. Floodfilling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. It is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap. **Note**: because the mask is larger than the filled image, a pixel in `mask` that corresponds to (x, y) pixel in `image` will have coordinates $(x + 1, y + 1)$

The function fills a connected component starting from the seed point with the specified color. The connectivity is determined by the closeness of pixel values. The pixel at (x, y) is considered to belong to the repainted domain if:

grayscale image, floating range

$$src(x', y') - \texttt{lo\_diff} <= src(x, y) <= src(x', y') + \texttt{up\_diff}$$

grayscale image, fixed range

$$src(seed.x, seed.y) - \texttt{lo\_diff} <= src(x, y) <= src(seed.x, seed.y) + \texttt{up\_diff}$$

color image, floating range

$$src(x', y')_r - \texttt{lo\_diff}_r <= src(x, y)_r <= src(x', y')_r + \texttt{up\_diff}_r$$
$$src(x', y')_g - \texttt{lo\_diff}_g <= src(x, y)_g <= src(x', y')_g + \texttt{up\_diff}_g$$
$$src(x', y')_b - \texttt{lo\_diff}_b <= src(x, y)_b <= src(x', y')_b + \texttt{up\_diff}_b$$

color image, fixed range

$$src(seed.x, seed.y)_r - \texttt{lo\_diff}_r <= src(x, y)_r <= src(seed.x, seed.y)_r + \texttt{up\_diff}_r$$
$$src(seed.x, seed.y)_g - \texttt{lo\_diff}_g <= src(x, y)_g <= src(seed.x, seed.y)_g + \texttt{up\_diff}_g$$
$$src(seed.x, seed.y)_b - \texttt{lo\_diff}_b <= src(x, y)_b <= src(seed.x, seed.y)_b + \texttt{up\_diff}_b$$

where $src(x', y')$ is the value of one of pixel neighbors. That is, to be added to the connected component, a pixel's color/brightness should be close enough to the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

cvInpaint

Inpaints the selected region in the image.

```
void cvInpaint(
           const CvArr* src,
           const CvArr* mask,
           CvArr* dst,
           double inpaintRadius,
           int flags);
```

src The input 8-bit 1-channel or 3-channel image.

mask The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.

dst The output image of the same format and the same size as input.

inpaintRadius The radius of circlular neighborhood of each point inpainted that is considered by the algorithm.

flags The inpainting method, one of the following:

CV_INPAINT_NS Navier-Stokes based method.

CV_INPAINT_TELEA The method by Alexandru Telea [**?**]

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video.

cvIntegral

Calculates the integral of an image.

```
void cvIntegral(
           const CvArr* image,
           CvArr* sum,
           CvArr* sqsum=NULL,
           CvArr* tiltedSum=NULL );
```

image The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

sum The integral image, $(W+1) \times (H+1)$, 32-bit integer or double precision floating-point (64f)

sqsum The integral image for squared pixel values, $(W+1) \times (H+1)$, double precision floating-point (64f)

tiltedSum The integral for the image rotated by 45 degrees, $(W+1) \times (H+1)$, the same data type as `sum`

The function calculates one or more integral images for the source image as following:

$$\texttt{sum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)$$

$$\texttt{sqsum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)^2$$

$$\texttt{tiltedSum}(X,Y) = \sum_{y<Y,abs(x-X)<y} \texttt{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum_{x_1<=x<x_2,\, y_1<=y<y_2} = \texttt{sum}(x_2,y_2) - \texttt{sum}(x_1,y_2) - \texttt{sum}(x_2,y_1) + \texttt{sum}(x_1,x_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

---

cvPyrMeanShiftFiltering

Does meanshift image segmentation

```
void cvPyrMeanShiftFiltering(
           const CvArr* src,
           CvArr* dst,
           double sp,
           double sr,
           int max_level=1,
           CvTermCriteria termcrit=
           cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,5,1));
```

src The source 8-bit, 3-channel image.

dst The destination image of the same format and the same size as the source.

sp The spatial window radius.

sr The color window radius.

max_level Maximum level of the pyramid for the segmentation.

termcrit Termination criteria: when to stop meanshift iterations.

The function implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X, Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X, Y) neighborhood in the joint space-color hyperspace is considered:

$$(x, y) : X - \texttt{sp} \leq x \leq X + \texttt{sp}, Y - \texttt{sp} \leq y \leq Y + \texttt{sp}, ||(R, G, B) - (r, g, b)|| \leq \texttt{sr}$$

where `(R,G,B)` and `(r,g,b)` are the vectors of color components at `(X,Y)` and `(x,y)`, respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value `(X',Y')` and average color vector `(R',G',B')` are found and they act as the neighborhood center on the next iteration:

(X, Y) $(X', Y'), (R, G, B)$ (R', G', B').

After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration):

$I(X, Y) < -(R*, G*, B*)$

Then $\texttt{max\_level} > 0$, the gaussian pyramid of $\texttt{max\_level} + 1$ levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much ($> \texttt{sr}$) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when $\texttt{max\_level} == 0$).

---

cvPyrSegmentation

Implements image segmentation by pyramids.

```
void cvPyrSegmentation(
           IplImage* src,
           IplImage* dst,
           CvMemStorage* storage,
           CvSeq** comp,
           int level,
           double threshold1,
           double threshold2 );
```

src The source image

dst The destination image

storage Storage; stores the resulting sequence of connected components

comp Pointer to the output sequence of the segmented components

level Maximum level of the pyramid for the segmentation

threshold1 Error threshold for establishing the links

threshold2 Error threshold for the segments clustering

The function implements image segmentation by pyramids. The pyramid builds up to the level `level`. The links between any pixel `a` on level `i` and its candidate father pixel `b` on the adjacent level are established if $p(c(a), c(b)) < threshold1$. After the connected components are defined, they are joined into several clusters. Any two segments A and B belong to the same cluster, if $p(c(A), c(B)) < threshold2$. If the input image has only one channel, then $p(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then

$$p(c^1, c^2) = 0.30(c^1_r - c^2_r) + 0.59(c^1_g - c^2_g) + 0.11(c^1_b - c^2_b).$$

There may be more than one connected component per a cluster. The images `src` and `dst` should be 8-bit single-channel or 3-channel images or equal size.

---

cvThreshold

Applies a fixed-level threshold to array elements.

```
double cvThreshold(
           const CvArr* src,
           CvArr* dst,
           double threshold,
           double maxValue,
           int thresholdType );
```

src Source array (single-channel, 8-bit or 32-bit floating point)

dst Destination array; must be either the same type as `src` or 8-bit

threshold Threshold value

maxValue Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types

thresholdType Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (cvCmpS could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by `thresholdType`:

CV_THRESH_BINARY

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxValue} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

CV_THRESH_TRUNC

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

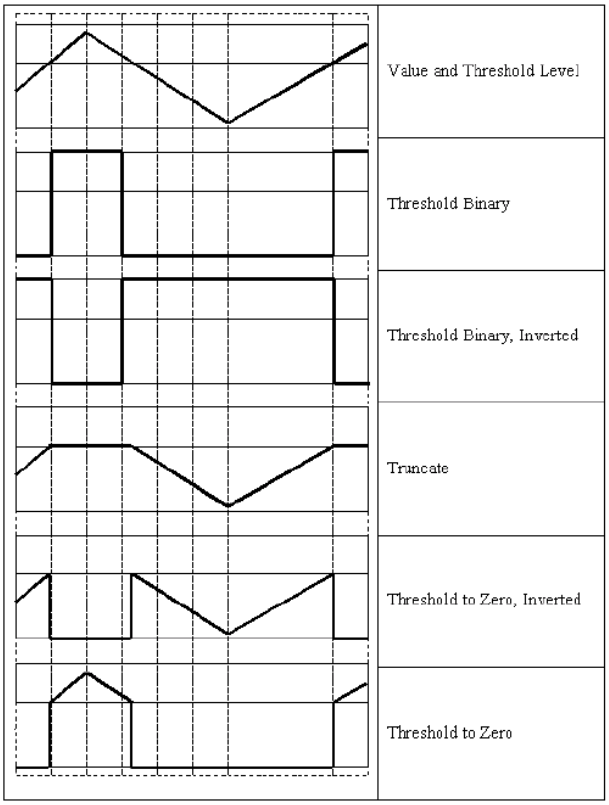
CV_THRESH_TOZERO

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_TOZERO_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Also, the special value CV_THRESH_OTSU may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified thresh. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.



2.4 Histograms

CvHistogram

Multi-dimensional histogram.

```
typedef struct CvHistogram
{
```

```
    int     type;
    CvArr*  bins;
    float   thresh[CV_MAX_DIM][2]; /* for uniform histograms */
    float** thresh2; /* for non-uniform histograms */
    CvMatND mat; /* embedded matrix header for array histograms */
}
CvHistogram;
```

---

cvCalcBackProject

Calculates the back projection.

```
void cvCalcBackProject(
           IplImage** image,
           CvArr* back_project,
           const CvHistogram* hist );
```

image Source images (though you may pass CvMat** as well)

back_project Destination back projection image of the same type as the source images

hist Histogram

The function calculates the back project of the histogram. For each tuple of pixels at the same position of all input single-channel images the function puts the value of the histogram bin, corresponding to the tuple in the destination image. In terms of statistics, the value of each output image pixel is the probability of the observed tuple given the distribution (histogram). For example, to find a red object in the picture, one may do the following:

1. Calculate a hue histogram for the red object assuming the image contains only this object. The histogram is likely to have a strong maximum, corresponding to red color.
2. Calculate back projection of a hue plane of input image where the object is searched, using the histogram. Threshold the image.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

That is the approximate algorithm of Camshift color object tracker, except for the 3rd step, instead of which CAMSHIFT algorithm is used to locate the object on the back projection given the previous object position.

cvCalcBackProjectPatch

Locates a template within an image by using a histogram comparison.

```
void cvCalcBackProjectPatch(
           IplImage** images,
           CvArr* dst,
           CvSize patch_size,
           CvHistogram* hist,
           int method,
           float factor );
```

images Source images (though, you may pass CvMat** as well)

dst Destination image

patch_size Size of the patch slid though the source image

hist Histogram

method Compasion method, passed to cvCompareHist (see description of that function)

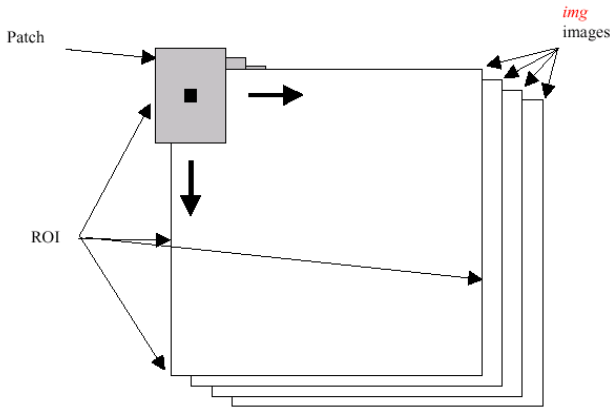
factor Normalization factor for histograms, will affect the normalization scale of the destination image, pass 1 if unsure

The function calculates the back projection by comparing histograms of the source image patches with the given histogram. Taking measurement results from some image at each location over ROI creates an array `image`. These results might be one or more of hue, `x` derivative, `y` derivative, Laplacian filter, oriented Gabor filter, etc. Each measurement output is collected into its own separate image. The `image` image array is a collection of these measurement images. A multi-dimensional histogram `hist` is constructed by sampling from the `image` image array. The final histogram is normalized. The `hist` histogram has as many dimensions as the number of elements in `image` array.

Each new image is measured and then converted into an `image` image array over a chosen ROI. Histograms are taken from this `image` image in an area covered by a "patch" with an anchor at center as shown in the picture below. The histogram is normalized using the parameter `norm_factor` so that it may be compared with `hist`. The calculated histogram is compared to the model histogram; `hist` uses The function `cvCompareHist` with the comparison method=`method`). The resulting output is placed at the location corresponding to the patch anchor in the probability image `dst`. This process is repeated as the patch is slid over the ROI.

Iterative histogram update by subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can save a lot of operations, though it is not implemented yet.

Back Project Calculation by Patches



---

cvCalcHist

Calculates the histogram of image(s).

```
void cvCalcHist(
           IplImage** image,
           CvHistogram* hist,
           int accumulate=0,
           const CvArr* mask=NULL );
```

image Source images (though you may pass CvMat** as well)

hist Pointer to the histogram

accumulate Accumulation flag. If it is set, the histogram is not cleared in the beginning. This feature allows user to compute a single histogram from several images, or to update the histogram online

mask The operation mask, determines what pixels of the source images are counted

The function calculates the histogram of one or more single-channel images. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input images.

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv )
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* v_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* planes[] = { h_plane, s_plane };
        IplImage* hsv = cvCreateImage( cvGetSize(src), 8, 3 );
        int h_bins = 30, s_bins = 32;
        int hist_size[] = {h_bins, s_bins};
        /* hue varies from 0 (~0 deg red) to 180 (~360 deg red again) */
        float h_ranges[] = { 0, 180 };
        /* saturation varies from 0 (black-gray-white) to
           255 (pure spectrum color) */
        float s_ranges[] = { 0, 255 };
        float* ranges[] = { h_ranges, s_ranges };
        int scale = 10;
        IplImage* hist_img =
            cvCreateImage( cvSize(h_bins*scale,s_bins*scale), 8, 3 );
        CvHistogram* hist;
        float max_value = 0;
        int h, s;

        cvCvtColor( src, hsv, CV_BGR2HSV );
        cvCvtPixToPlane( hsv, h_plane, s_plane, v_plane, 0 );
        hist = cvCreateHist( 2, hist_size, CV_HIST_ARRAY, ranges, 1 );
        cvCalcHist( planes, hist, 0, 0 );
        cvGetMinMaxHistValue( hist, 0, &max_value, 0, 0 );
        cvZero( hist_img );

        for( h = 0; h < h_bins; h++ )
        {
            for( s = 0; s < s_bins; s++ )
            {
                float bin_val = cvQueryHistValue\_2D( hist, h, s );
                int intensity = cvRound(bin_val*255/max_value);
                cvRectangle( hist_img, cvPoint( h*scale, s*scale ),
                             cvPoint( (h+1)*scale - 1, (s+1)*scale - 1),
                             CV_RGB(intensity,intensity,intensity),
                             CV_FILLED );
```

```
            }
        }

        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "H-S Histogram", 1 );
        cvShowImage( "H-S Histogram", hist_img );

        cvWaitKey(0);
    }
}
```

---

cvCalcProbDensity

Divides one histogram by another.

```
void cvCalcProbDensity(
           const CvHistogram* hist1,
           const CvHistogram* hist2,
           CvHistogram* dst_hist,
           double scale=255 );
```

hist1 first histogram (the divisor)

hist2 second histogram

dst_hist destination histogram

scale scale factor for the destination histogram

The function calculates the object probability density from the two histograms as:

$$\texttt{dist\_hist}(I) = \begin{cases} 0 & \text{if } \texttt{hist1}(I) = 0 \\ \texttt{scale} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) > \texttt{hist1}(I) \\ \frac{\texttt{hist2}(I) \cdot \texttt{scale}}{\texttt{hist1}(I)} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) \leq \texttt{hist1}(I) \end{cases}$$

So the destination histogram bins are within less than `scale`.

cvClearHist

Clears the histogram.

```
void cvClearHist( CvHistogram* hist );
```

hist Histogram

The function sets all of the histogram bins to 0 in the case of a dense histogram and removes all histogram bins in the case of a sparse array.

cvCompareHist

Compares two dense histograms.

```
double cvCompareHist(
           const CvHistogram* hist1,
           const CvHistogram* hist2,
           int method );
```

hist1 The first dense histogram

hist2 The second dense histogram

method Comparison method, one of the following:

CV_COMP_CORREL Correlation

CV_COMP_CHISQR Chi-Square

CV_COMP_INTERSECT Intersection

CV_COMP_BHATTACHARYYA Bhattacharyya distance

The function compares two dense histograms using the specified method (H_1 denotes the first histogram, H_2 the second):

Correlation (method=CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H_1'(I) \cdot H_2'(I))}{\sqrt{\sum_I (H_1'(I)^2) \cdot \sum_I (H_2'(I)^2)}}$$

where

$$H'_k(I) = \frac{H_k(I) - 1}{N \cdot \sum_J H_k(J)}$$

where N is the number of histogram bins.

Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \sum_I \frac{\sqrt{H_1(I) \cdot H_2(I)}}{\sqrt{\sum_I H_1(I) \cdot \sum_I H_2(I)}}}$$

The function returns $d(H_1, H_2)$.

Note: the method `CV_COMP_BHATTACHARYYA` only works with normalized histograms.

To compare a sparse histogram or more general sparse configurations of weighted points, consider using the cvCalcEMD2 function.

---

cvCopyHist

Copies a histogram.

```
void cvCopyHist( const CvHistogram* src, CvHistogram** dst );
```

src Source histogram

dst Pointer to destination histogram

The function makes a copy of the histogram. If the second histogram pointer `*dst` is NULL, a new histogram of the same size as `src` is created. Otherwise, both histograms must have equal types and sizes. Then the function copies the source histogram's bin values to the destination histogram and sets the same bin value ranges as in `src`.

cvCreateHist

Creates a histogram.

```
CvHistogram* cvCreateHist(
            int dims,
            int* sizes,
            int type,
            float** ranges=NULL,
            int uniform=1 );
```

dims Number of histogram dimensions

sizes Array of the histogram dimension sizes

type Histogram representation format: `CV_HIST_ARRAY` means that the histogram data is represented as a multi-dimensional dense array CvMatND; `CV_HIST_SPARSE` means that histogram data is represented as a multi-dimensional sparse array CvSparseMat

ranges Array of ranges for the histogram bins. Its meaning depends on the `uniform` parameter value. The ranges are used for when the histogram is calculated or backprojected to determine which histogram bin corresponds to which value/tuple of values from the input image(s)

uniform Uniformity flag; if not 0, the histogram has evenly spaced bins and for every $0 <= i < cDims$ `ranges[i]` is an array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is then split into `dims[i]` equal parts to determine the `i-th` input tuple value ranges for every histogram bin. And if `uniform=0`, then `i-th` element of `ranges` array contains `dims[i]+1` elements: $\texttt{lower}_0, \texttt{upper}_0, \texttt{lower}_1, \texttt{upper}_1 = \texttt{lower}_2, ...\texttt{upper}_{dims[i]-1}$ where $\texttt{lower}_j$ and $\texttt{upper}_j$ are lower and upper boundaries of `i-th` input tuple value for `j-th` bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin are not counted by cvCalcHist and filled with 0 by cvCalcBackProject

The function creates a histogram of the specified size and returns a pointer to the created histogram. If the array `ranges` is 0, the histogram bin ranges must be specified later via the function cvSetHistBinRanges. Though cvCalcHist and cvCalcBackProject may process 8-bit images without setting bin ranges, they assume thy are equally spaced in 0 to 255 bins.

cvGetHistValue*D

Returns a pointer to the histogram bin.

```
float cvGetHistValue_1D(hist, idx0)
float cvGetHistValue_2D(hist, idx0, idx1)
float cvGetHistValue_3D(hist, idx0, idx1, idx2)
float cvGetHistValue_nD(hist, idx)
```

hist Histogram

idx0, idx1, idx2, idx3 Indices of the bin

idx Array of indices

```
#define cvGetHistValue_1D( hist, idx0 )
    ((float*)(cvPtr1D( (hist)->bins, (idx0), 0 ))
#define cvGetHistValue_2D( hist, idx0, idx1 )
    ((float*)(cvPtr2D( (hist)->bins, (idx0), (idx1), 0 )))
#define cvGetHistValue_3D( hist, idx0, idx1, idx2 )
    ((float*)(cvPtr3D( (hist)->bins, (idx0), (idx1), (idx2), 0 )))
#define cvGetHistValue_nD( hist, idx )
    ((float*)(cvPtrND( (hist)->bins, (idx), 0 )))
```

The macros `GetHistValue` return a pointer to the specified bin of the 1D, 2D, 3D or N-D histogram. In the case of a sparse histogram the function creates a new bin and sets it to 0, unless it exists already.

cvGetMinMaxHistValue

Finds the minimum and maximum histogram bins.

```
void cvGetMinMaxHistValue(
           const CvHistogram* hist,
           float* min_value,
           float* max_value,
           int* min_idx=NULL,
           int* max_idx=NULL );
```

hist Histogram

min_value Pointer to the minimum value of the histogram

max_value Pointer to the maximum value of the histogram

min_idx Pointer to the array of coordinates for the minimum

max_idx Pointer to the array of coordinates for the maximum

The function finds the minimum and maximum histogram bins and their positions. All of output arguments are optional. Among several extremas with the same value the ones with the minimum index (in lexicographical order) are returned. In the case of several maximums or minimums, the earliest in lexicographical order (extrema locations) is returned.

---

cvMakeHistHeaderForArray

Makes a histogram out of an array.

```
CvHistogram* cvMakeHistHeaderForArray(
            int dims,
            int* sizes,
            CvHistogram* hist,
            float* data,
            float** ranges=NULL,
            int uniform=1 );
```

dims Number of histogram dimensions

sizes Array of the histogram dimension sizes

hist The histogram header initialized by the function

data Array that will be used to store histogram bins

ranges Histogram bin ranges, see cvCreateHist

uniform Uniformity flag, see cvCreateHist

The function initializes the histogram, whose header and bins are allocated by th user. cvReleaseHist does not need to be called afterwards. Only dense histograms can be initialized this way. The function returns `hist`.

cvNormalizeHist

Normalizes the histogram.

```
void cvNormalizeHist( CvHistogram* hist, double factor );
```

hist Pointer to the histogram

factor Normalization factor

The function normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor`.

cvQueryHistValue*D

Queries the value of the histogram bin.

```
float QueryHistValue_1D(CvHistogram hist, int idx0)
```

hist Histogram

idx0, idx1, idx2, idx3 Indices of the bin

idx Array of indices

```
#define cvQueryHistValue\_1D( hist, idx0 ) \
    cvGetReal1D( (hist)->bins, (idx0) )
#define cvQueryHistValue\_2D( hist, idx0, idx1 ) \
    cvGetReal2D( (hist)->bins, (idx0), (idx1) )
#define cvQueryHistValue\_3D( hist, idx0, idx1, idx2 ) \
    cvGetReal3D( (hist)->bins, (idx0), (idx1), (idx2) )
#define cvQueryHistValue\_nD( hist, idx ) \
    cvGetRealND( (hist)->bins, (idx) )
```

The macros return the value of the specified bin of the 1D, 2D, 3D or N-D histogram. In the case of a sparse histogram the function returns 0, if the bin is not present in the histogram no new bin is created.

cvReleaseHist

Releases the histogram.

```
void cvReleaseHist( CvHistogram** hist );
```

hist Double pointer to the released histogram

The function releases the histogram (header and the data). The pointer to the histogram is cleared by the function. If `*hist` pointer is already `NULL`, the function does nothing.

cvSetHistBinRanges

Sets the bounds of the histogram bins.

```
void cvSetHistBinRanges(
           CvHistogram* hist,
           float** ranges,
           int uniform=1 );
```

hist Histogram

ranges Array of bin ranges arrays, see cvCreateHist

uniform Uniformity flag, see cvCreateHist

The function is a stand-alone function for setting bin ranges in the histogram. For a more detailed description of the parameters `ranges` and `uniform` see the cvCalcHist function, that can initialize the ranges as well. Ranges for the histogram bins must be set before the histogram is calculated or the backproject of the histogram is calculated.

cvThreshHist

Thresholds the histogram.

```
void cvThreshHist( CvHistogram* hist, double threshold );
```

hist Pointer to the histogram

threshold Threshold level

The function clears histogram bins that are below the specified threshold.

2.5 Feature Detection

---

cvCanny

Implements the Canny algorithm for edge detection.

```
void cvCanny(
           const CvArr* image,
           CvArr* edges,
           double threshold1,
           double threshold2,
           int aperture_size=3 );
```

image Single-channel input image

edges Single-channel image to store the edges found by the function

threshold1 The first threshold

threshold2 The second threshold

aperture_size Aperture parameter for the Sobel operator (see cvSobel)

The function finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges.

cvCornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
void cvCornerEigenValsAndVecs(
           const CvArr* image,
           CvArr* eigenvv,
           int blockSize,
           int aperture_size=3 );
```

image Input image

eigenvv Image to store the results. It must be 6 times wider than the input image

blockSize Neighborhood size (see discussion)

aperture_size Aperture parameter for the Sobel operator (see cvSobel)

For every pixel, the function `cvCornerEigenValsAndVecs` considers a `blockSize×blockSize` neigborhood S(p). It calcualtes the covariation matrix of derivatives over the neigborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dx \cdot dI/dy)^2 \\ \sum_{S(p)}(dI/dx \cdot dI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the eigenvalues of M; not sorted

x_1, y_1 are the eigenvectors corresponding to λ_1

x_2, y_2 are the eigenvectors corresponding to λ_2

cvCornerHarris

Harris edge detector.

```
void cvCornerHarris(
           const CvArr* image,
           CvArr* harris_dst,
           int blockSize,
           int aperture_size=3,
           double k=0.04 );
```

image Input image

harris_dst Image to store the Harris detector responses. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cvCornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cvSobel).

k Harris detector free parameter. See the formula below

The function runs the Harris edge detector on the image. Similarly to cvCornerMinEigenVal and cvCornerEigenValsAndVecs, for each pixel it calculates a 2×2 gradient covariation matrix M over a `blockSize` $\times$ `blockSize` neighborhood. Then, it stores

$$det(M) - k\, trace(M)^2$$

to the destination image. Corners in the image can be found as the local maxima of the destination image.

---

cvCornerMinEigenVal

Calculates the minimal eigenvalue of gradient matrices for corner detection.

```
void cvCornerMinEigenVal(
           const CvArr* image,
           CvArr* eigenval,
           int blockSize,
           int aperture_size=3 );
```

image Input image

eigenval Image to store the minimal eigenvalues. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cvCornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cvSobel).

The function is similar to cvCornerEigenValsAndVecs but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. $min(\lambda_1, \lambda_2)$ in terms of the previous function.

---

cvExtractSURF

Extracts Speeded Up Robust Features from an image.

```
void cvExtractSURF(
            const CvArr* image,
            const CvArr* mask,
            CvSeq** keypoints,
            CvSeq** descriptors,
            CvMemStorage* storage,
            CvSURFParams params );
```

image The input 8-bit grayscale image

mask The optional input 8-bit mask. The features are only found in the areas that contain more than 50% of non-zero mask pixels

keypoints The output parameter; double pointer to the sequence of keypoints. The sequence of CvSURFPoint structures is as follows:

```
typedef struct CvSURFPoint
{
   CvPoint2D32f pt; // position of the feature within the image
   int laplacian;   // -1, 0 or +1. sign of the laplacian at the point.
                    // can be used to speedup feature comparison
                    // (normally features with laplacians of different
            // signs can not match)
   int size;        // size of the feature
   float dir;       // orientation of the feature: 0..360 degrees
   float hessian;   // value of the hessian (can be used to
            // approximately estimate the feature strengths;
                    // see also params.hessianThreshold)
}
CvSURFPoint;
```

descriptors The optional output parameter; double pointer to the sequence of descriptors. Depending on the params.extended value, each element of the sequence will be either a 64-element or a 128-element floating-point (CV_32F) vector. If the parameter is NULL, the descriptors are not computed

storage Memory storage where keypoints and descriptors will be stored

params Various algorithm parameters put to the structure CvSURFParams:

```
typedef struct CvSURFParams
{
   int extended; // 0 means basic descriptors (64 elements each),
                 // 1 means extended descriptors (128 elements each)
   double hessianThreshold; // only features with keypoint.hessian
         // larger than that are extracted.
                 // good default value is ~300-500 (can depend on the
         // average local contrast and sharpness of the image).
                 // user can further filter out some features based on
         // their hessian values and other characteristics.
   int nOctaves; // the number of octaves to be used for extraction.
                 // With each next octave the feature size is doubled
         // (3 by default)
   int nOctaveLayers; // The number of layers within each octave
         // (4 by default)
}
CvSURFParams;

CvSURFParams cvSURFParams(double hessianThreshold, int extended=0);
         // returns default parameters
```

The function cvExtractSURF finds robust features in the image, as described in Bay06 . For each feature it returns its location, size, orientation and optionally the descriptor, basic or extended. The function can be used for object tracking and localization, image stitching etc. See the find_obj.cpp demo in OpenCV samples directory.

---

cvFindCornerSubPix

Refines the corner locations.

```
void cvFindCornerSubPix(
           const CvArr* image,
           CvPoint2D32f* corners,
```

```
            int count,
            CvSize win,
            CvSize zero_zone,
            CvTermCriteria criteria );
```

image Input image

corners Initial coordinates of the input corners; refined coordinates on output

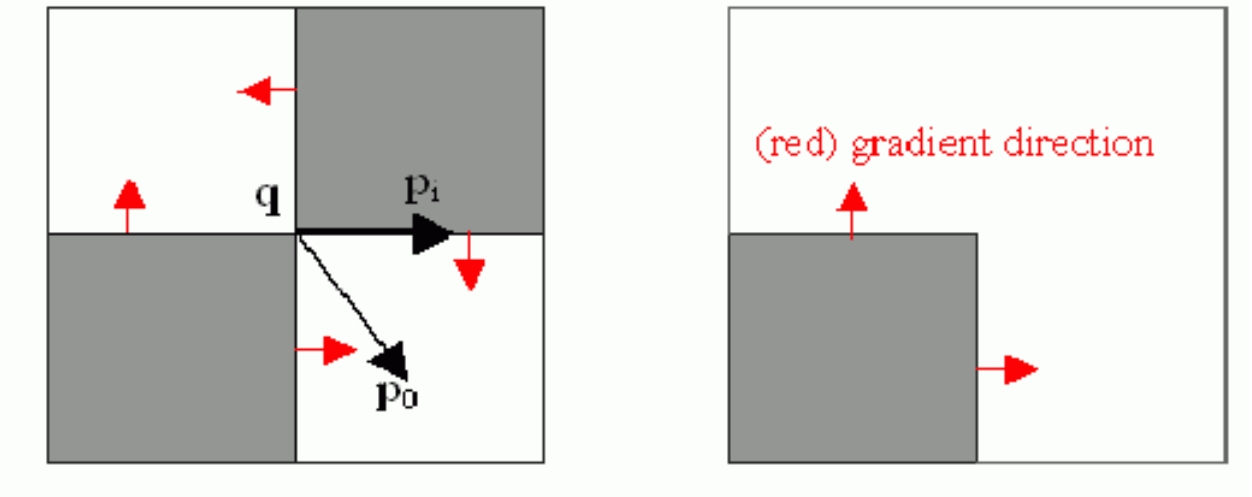
count Number of corners

win Half of the side length of the search window. For example, if `win`=(5,5), then a $5*2+1 \times 5*2+1 = 11 \times 11$ search window would be used

zero_zone Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size

criteria Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below.



Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider the expression:

$$\epsilon_i = {DI_{p_i}}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot {DI_{p_i}}^T) q = \sum_i (DI_{p_i} \cdot {DI_{p_i}}^T \cdot p_i)$$

where the gradients are summed within a neighborhood ("search window") of q. Calling the first gradient term G and the second gradient term b gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center q and then iterates until the center keeps within a set threshold.

---

cvGetStarKeypoints

Retrieves keypoints using the StarDetector algorithm.

```
CvSeq* cvGetStarKeypoints(
           const CvArr* image,
           CvMemStorage* storage,
           CvStarDetectorParams params=cvStarDetectorParams() );
```

image The input 8-bit grayscale image

storage Memory storage where the keypoints will be stored

params Various algorithm parameters given to the structure CvStarDetectorParams:

```
typedef struct CvStarDetectorParams
{
   int maxSize; // maximal size of the features detected. The following
                // values of the parameter are supported:
                // 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
   int responseThreshold; // threshold for the approximatd laplacian,
```

```
                            // used to eliminate weak features
    int lineThresholdProjected; // another threshold for laplacian to
                // eliminate edges
    int lineThresholdBinarized; // another threshold for the feature
                // scale to eliminate edges
    int suppressNonmaxSize; // linear size of a pixel neighborhood
                // for non-maxima suppression
}
CvStarDetectorParams;
```

The function GetStarKeypoints extracts keypoints that are local scale-space extremas. The scale-space is constructed by computing approximate values of laplacians with different sigma's at each pixel. Instead of using pyramids, a popular approach to save computing time, all of the laplacians are computed at each pixel of the original high-resolution image. But each approximate laplacian value is computed in O(1) time regardless of the sigma, thanks to the use of integral images. The algorithm is based on the paper Agrawal08 , but instead of a square, hexagon or octagon it uses an 8-end star shape, hence the name, consisting of overlapping upright and tilted squares.

Each computed feature is represented by the following structure:

```
typedef struct CvStarKeypoint
{
    CvPoint pt; // coordinates of the feature
    int size; // feature size, see CvStarDetectorParams::maxSize
    float response; // the approximated laplacian value at that point.
}
CvStarKeypoint;

inline CvStarKeypoint cvStarKeypoint(CvPoint pt, int size, float response);
```

Below is the small usage sample:

```
#include "cv.h"
#include "highgui.h"

int main(int argc, char** argv)
{
    const char* filename = argc > 1 ? argv[1] : "lena.jpg";
    IplImage* img = cvLoadImage( filename, 0 ), *cimg;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* keypoints = 0;
    int i;

    if( !img )
        return 0;
```

```
    cvNamedWindow( "image", 1 );
    cvShowImage( "image", img );
    cvNamedWindow( "features", 1 );
    cimg = cvCreateImage( cvGetSize(img), 8, 3 );
    cvCvtColor( img, cimg, CV_GRAY2BGR );

    keypoints = cvGetStarKeypoints( img, storage, cvStarDetectorParams(45) );

    for( i = 0; i < (keypoints ? keypoints->total : 0); i++ )
    {
        CvStarKeypoint kpt = *(CvStarKeypoint*)cvGetSeqElem(keypoints, i);
        int r = kpt.size/2;
        cvCircle( cimg, kpt.pt, r, CV_RGB(0,255,0));
        cvLine( cimg, cvPoint(kpt.pt.x + r, kpt.pt.y + r),
            cvPoint(kpt.pt.x - r, kpt.pt.y - r), CV_RGB(0,255,0));
        cvLine( cimg, cvPoint(kpt.pt.x - r, kpt.pt.y + r),
            cvPoint(kpt.pt.x + r, kpt.pt.y - r), CV_RGB(0,255,0));
    }
    cvShowImage( "features", cimg );
    cvWaitKey();
}
```

---

cvGoodFeaturesToTrack

Determines strong corners on an image.

```
void cvGoodFeaturesToTrack(
           const CvArr* image
           CvArr* eigImage, CvArr* tempImage
           CvPoint2D32f* corners
           int* cornerCount
           double qualityLevel
           double minDistance
           const CvArr* mask=NULL
           int blockSize=3
           int useHarris=0
           double k=0.04 );
```

image The source 8-bit or floating-point 32-bit, single-channel image

eigImage Temporary floating-point 32-bit image, the same size as `image`

tempImage Another temporary image, the same size and format as `eigImage`

corners Output parameter; detected corners

cornerCount Output parameter; number of detected corners

qualityLevel Multiplier for the max/min eigenvalue; specifies the minimal accepted quality of image corners

minDistance Limit, specifying the minimum possible distance between the returned corners; Euclidian distance is used

mask Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL

blockSize Size of the averaging block, passed to the underlying cvCornerMinEigenVal or cvCornerHarris used by the function

useHarris If nonzero, Harris operator (cvCornerHarris) is used instead of default cvCornerMinEigenVal

k Free parameter of Harris detector; used only if ($\texttt{useHarris} \neq 0$)

The function finds the corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using the cvCornerMinEigenVal function and stores them in `eigImage`. Then it performs non-maxima suppression (only the local maxima in 3×3 neighborhood are retained). The next step rejects the corners with the minimal eigenvalue less than $\texttt{qualityLevel} \cdot max(\texttt{eigImage}(x, y))$. Finally, the function ensures that the distance between any two corners is not smaller than `minDistance`. The weaker corners (with a smaller min eigenvalue) that are too close to the stronger corners are rejected.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel`, and `A` ¿ B, the array of returned corners with `qualityLevel=A` will be the prefix of the output corners array with `qualityLevel=B`.

---

cvHoughLines2

Finds lines in a binary image using a Hough transform.

```
CvSeq* cvHoughLines2(
           CvArr* image,
           void* storage,
           int method,
           double rho,
           double theta,
           int threshold,
           double param1=0,
           double param2=0 );
```

image The 8-bit, single-channel, binary source image. In the case of a probabilistic method, the image is modified by the function

storage The storage for the lines that are detected. It can be a memory storage (in this case a sequence of lines is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of a particular type (see below) to which the lines' parameters are written. The matrix header is modified by the function so its `cols` or `rows` will contain the number of lines detected. If `storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (in the case of standard hough transform the lines are sorted by the accumulator value)

method The Hough transform variant, one of the following:

- **CV_HOUGH_STANDARD** classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ), where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of `CV_32FC2` type
- **CV_HOUGH_PROBABILISTIC** probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole line. Each segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of `CV_32SC4` type
- **CV_HOUGH_MULTI_SCALE** multi-scale variant of the classical Hough transform. The lines are encoded the same way as `CV_HOUGH_STANDARD`

rho Distance resolution in pixel-related units

theta Angle resolution measured in radians

threshold Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`

param1 The first method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the minimum line length.
- For the multi-scale Hough transform it is the divisor for the distance resolution ρ. (The coarse distance resolution will be ρ and the accurate resolution will be $(\rho/\texttt{param1})$).

param2 The second method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the maximum gap between line segments lying on the same line to treat them as a single line segment (i.e. to join them).
- For the multi-scale Hough transform it is the divisor for the angle resolution θ. (The coarse angle resolution will be θ and the accurate resolution will be $(\theta/\texttt{param2})$).

The function implements a few variants of the Hough transform for line detection.

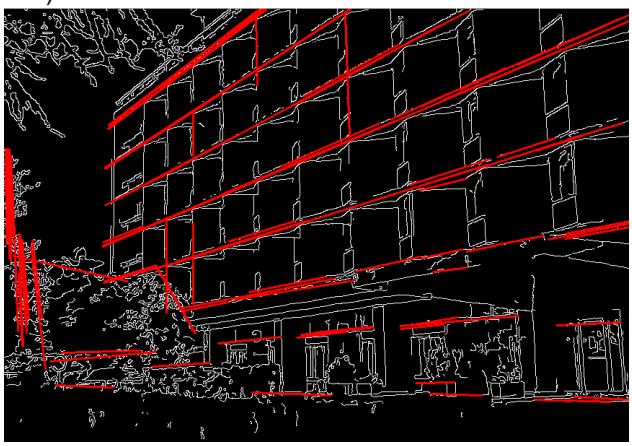
Example. Detecting lines with Hough transform.

```
/* This is a standalone program. Pass an image name as a first parameter
of the program.  Switch between standard and probabilistic Hough transform
by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* lines = 0;
        int i;
        cvCanny( src, dst, 50, 200, 3 );
        cvCvtColor( dst, color_dst, CV_GRAY2BGR );
#if 1
        lines = cvHoughLines2( dst,
                               storage,
                               CV_HOUGH_STANDARD,
                               1,
                               CV_PI/180,
```

```
                            100,
                            0,
                            0 );

        for( i = 0; i < MIN(lines->total,100); i++ )
        {
            float* line = (float*)cvGetSeqElem(lines,i);
            float rho = line[0];
            float theta = line[1];
            CvPoint pt1, pt2;
            double a = cos(theta), b = sin(theta);
            double x0 = a*rho, y0 = b*rho;
            pt1.x = cvRound(x0 + 1000*(-b));
            pt1.y = cvRound(y0 + 1000*(a));
            pt2.x = cvRound(x0 - 1000*(-b));
            pt2.y = cvRound(y0 - 1000*(a));
            cvLine( color_dst, pt1, pt2, CV_RGB(255,0,0), 3, 8 );
        }
#else
        lines = cvHoughLines2( dst,
                            storage,
                            CV_HOUGH_PROBABILISTIC,
                            1,
                            CV_PI/180,
                            80,
                            30,
                            10 );
        for( i = 0; i < lines->total; i++ )
        {
            CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
            cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3, 8 );
        }
#endif
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "Hough", 1 );
        cvShowImage( "Hough", color_dst );

        cvWaitKey(0);
    }
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in the case of probabilistic Hough transform (`#if 0` case):



---

cvPreCornerDetect

Calculates the feature map for corner detection.

```
void cvPreCornerDetect(
           const CvArr* image,
           CvArr* corners,
           int apertureSize=3 );
```

image Input image

corners Image to store the corner candidates

apertureSize Aperture parameter for the Sobel operator (see cvSobel)

The function calculates the function

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}$$

where $D_?$ denotes one of the first image derivatives and $D_{??}$ denotes a second image derivative.

The corners can be found as local maximums of the function below:

```
// assume that the image is floating-point
IplImage* corners = cvCloneImage(image);
IplImage* dilated_corners = cvCloneImage(image);
IplImage* corner_mask = cvCreateImage( cvGetSize(image), 8, 1 );
cvPreCornerDetect( image, corners, 3 );
cvDilate( corners, dilated_corners, 0, 1 );
cvSubS( corners, dilated_corners, corners );
cvCmpS( corners, 0, corner_mask, CV_CMP_GE );
cvReleaseImage( &corners );
cvReleaseImage( &dilated_corners );
```

---

cvSampleLine

Reads the raster line to the buffer.

```
int cvSampleLine(
           const CvArr* image
           CvPoint pt1
           CvPoint pt2
           void* buffer
           int connectivity=8 );
```

image Image to sample the line from

pt1 Starting line point

pt2 Ending line point

buffer Buffer to store the line points; must have enough size to store $max(|\texttt{pt2.x} - \texttt{pt1.x}| + 1, |\texttt{pt2.y} - \texttt{pt1.y}| + 1)$ points in the case of an 8-connected line and $(|\texttt{pt2.x} - \texttt{pt1.x}| + |\texttt{pt2.y} - \texttt{pt1.y}| + 1)$ in the case of a 4-connected line

connectivity The line connectivity, 4 or 8

The function implements a particular application of line iterators. The function reads all of the image points lying on the line between `pt1` and `pt2`, including the end points, and stores them into the buffer.

2.6 Motion Analysis and Object Tracking

---

cvAcc

Adds a frame to an accumulator.

```
void cvAcc(
           const CvArr* image,
           CvArr* sum,
           const CvArr* mask=NULL );
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently)

sum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the whole image `image` or its selected region to the accumulator `sum`:

$$\texttt{sum}(x,y) \leftarrow \texttt{sum}(x,y) + \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cvCalcGlobalOrientation

Calculates the global motion orientation of some selected region.

```
double cvCalcGlobalOrientation(
           const CvArr* orientation,
           const CvArr* mask,
           const CvArr* mhi,
           double timestamp,
           double duration );
```

orientation Motion gradient orientation image; calculated by the function cvCalcMotionGradient

mask Mask image. It may be a conjunction of a valid gradient mask, obtained with cvCalcMotionGradient and the mask of the region, whose direction needs to be calculated

mhi Motion history image

timestamp Current time in milliseconds or other units, it is better to store time passed to cvUpdateMotionHistory before and reuse it here, because running cvUpdateMotionHistory and cvCalcMotionGradient on large images may take some time

duration Maximal duration of motion track in milliseconds, the same as cvUpdateMotionHistory

The function calculates the general motion direction in the selected region and returns the angle between 0 degrees and 360 degrees . At first the function builds the orientation histogram and finds the basic orientation as a coordinate of the histogram maximum. After that the function calculates the shift relative to the basic orientation as a weighted sum of all of the orientation vectors: the more recent the motion, the greater the weight. The resultant angle is a circular sum of the basic orientation and the shift.

---

cvCalcMotionGradient

Calculates the gradient orientation of a motion history image.

```
void cvCalcMotionGradient(
           const CvArr* mhi,
           CvArr* mask,
           CvArr* orientation,
           double delta1,
           double delta2,
           int apertureSize=3 );
```

mhi Motion history image

mask Mask image; marks pixels where the motion gradient data is correct; output parameter

orientation Motion gradient orientation image; contains angles from 0 to 360 degrees

delta1 See below

delta2 See below

apertureSize Aperture size of derivative operators used by the function: CV_SCHARR, 1, 3, 5 or 7 (see cvSobel)

The function calculates the derivatives Dx and Dy of `mhi` and then calculates gradient orientation as:

$$\texttt{orientation}(x,y) = \arctan \frac{Dy(x,y)}{Dx(x,y)}$$

where both $Dx(x,y)$ and $Dy(x,y)$ signs are taken into account (as in the cvCartToPolar function). After that `mask` is filled to indicate where the orientation is valid (see the `delta1` and `delta2` description).

The function finds the minimum ($m(x,y)$) and maximum ($M(x,y)$) mhi values over each pixel (x,y) neighborhood and assumes the gradient is valid only if

$$\min(\texttt{delta1}, \texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1}, \texttt{delta2}).$$

---

cvCalcOpticalFlowBM

Calculates the optical flow for two images by using the block matching method.

```
void cvCalcOpticalFlowBM(
           const CvArr* prev,
           const CvArr* curr,
           CvSize blockSize,
           CvSize shiftSize,
           CvSize max_range,
           int usePrevious,
           CvArr* velx,
           CvArr* vely );
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

blockSize Size of basic blocks that are compared

shiftSize Block coordinate increments

max_range Size of the scanned neighborhood in pixels around the block

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of

$$\left\lfloor \frac{\texttt{prev->width} - \texttt{blockSize.width}}{\texttt{shiftSize.width}} \right\rfloor \times \left\lfloor \frac{\texttt{prev->height} - \texttt{blockSize.height}}{\texttt{shiftSize.height}} \right\rfloor$$

size, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size `velx`, 32-bit floating-point, single-channel

The function calculates the optical flow for overlapped blocks `blockSize.width×blockSize.height` pixels each, thus the velocity fields are smaller than the original images. For every block in `prev` the functions tries to find a similar block in `curr` in some neighborhood of the original block or shifted by (velx(x0,y0),vely(x0,y0)) block as has been calculated by previous function call (if `usePrevious=1`)

---

cvCalcOpticalFlowHS

Calculates the optical flow for two images.

```
void cvCalcOpticalFlowHS(
           const CvArr* prev,
           const CvArr* curr,
           int usePrevious,
           CvArr* velx,
           CvArr* vely,
           double lambda,
           CvTermCriteria criteria );
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

lambda Lagrangian multiplier

criteria Criteria of termination of velocity computing

The function computes the flow for every pixel of the first input image using the Horn and Schunck algorithm [**?**].

---

cvCalcOpticalFlowLK

Calculates the optical flow for two images.

```
void cvCalcOpticalFlowLK(
           const CvArr* prev,
           const CvArr* curr,
           CvSize winSize,
           CvArr* velx,
           CvArr* vely );
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

winSize Size of the averaging window used for grouping pixels

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

The function computes the flow for every pixel of the first input image using the Lucas and Kanade algorithm [**?**].

cvCalcOpticalFlowPyrLK

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

```
void cvCalcOpticalFlowPyrLK(
           const CvArr* prev,
           const CvArr* curr,
           CvArr* prevPyr,
           CvArr* currPyr,
           const CvPoint2D32f* prevFeatures,
           CvPoint2D32f* currFeatures,
           int count,
           CvSize winSize,
           int level,
           char* status,
           float* track_error,
           CvTermCriteria criteria,
           int flags );
```

prev First frame, at time `t`

curr Second frame, at time `t + dt`

prevPyr Buffer for the pyramid for the first frame. If the pointer is not `NULL` , the buffer must have a sufficient size to store the pyramid from level `1` to level `level` ; the total size of `(image_width+8)*image_height/3` bytes is sufficient

currPyr Similar to `prevPyr`, used for the second frame

prevFeatures Array of points for which the flow needs to be found

currFeatures Array of 2D points containing the calculated new positions of the input features in the second image

count Number of feature points

winSize Size of the search window of each pyramid level

level Maximal pyramid level number. If `0` , pyramids are not used (single level), if `1` , two levels are used, etc

status Array. Every element of the array is set to `1` if the flow for the corresponding feature has been found, `0` otherwise

track_error Array of double numbers containing the difference between patches around the original and moved points. Optional parameter; can be `NULL`

criteria Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped

flags Miscellaneous flags:

CV_LKFLOWPyr_A_READY pyramid for the first frame is precalculated before the call

CV_LKFLOWPyr_B_READY pyramid for the second frame is precalculated before the call

CV_LKFLOW_INITIAL_GUESSES array B contains initial coordinates of features before the function call

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids [**?**] . It calculates the coordinates of the feature points on the current video frame given their coordinates on the previous frame. The function finds the coordinates with sub-pixel accuracy.

Both parameters `prevPyr` and `currPyr` comply with the following rules: if the image pointer is 0, the function allocates the buffer internally, calculates the pyramid, and releases the buffer after processing. Otherwise, the function calculates the pyramid and stores it in the buffer unless the flag `CV_LKFLOWPyr_A[B]_READY` is set. The image should be large enough to fit the Gaussian pyramid data. After the function call both pyramids are calculated and the readiness flag for the corresponding image can be set in the next call (i.e., typically, for all the image pairs except the very first one `CV_LKFLOWPyr_A_READY` is set).

---

cvCamShift

Finds the object center, size, and orientation.

```
int cvCamShift(
           const CvArr* prob_image,
           CvRect window,
           CvTermCriteria criteria,
           CvConnectedComp* comp,
           CvBox2D* box=NULL );
```

prob_image Back projection of object histogram (see cvCalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

box Circumscribed box for the object. If not `NULL`, it contains object size and orientation

The function implements the CAMSHIFT object tracking algrorithm [**?**]. First, it finds an object center using cvMeanShift and, after that, calculates the object size and orientation. The function returns number of iterations made within cvMeanShift.

The `CamShiftTracker` class declared in cv.hpp implements the color object tracker that uses the function.

CvConDensation

ConDenstation state.

```
typedef struct CvConDensation
{
    int MP;     //Dimension of measurement vector
    int DP;     // Dimension of state vector
    float* DynamMatr;       // Matrix of the linear Dynamics system
    float* State;           // Vector of State
    int SamplesNum;         // Number of the Samples
    float** flSamples;      // array of the Sample Vectors
    float** flNewSamples;   // temporary array of the Sample Vectors
    float* flConfidence;    // Confidence for each Sample
    float* flCumulative;    // Cumulative confidence
    float* Temp;            // Temporary vector
    float* RandomSample;    // RandomVector to update sample set
    CvRandState* RandS;     // Array of structures to generate random vectors
} CvConDensation;
```

The structure `CvConDensation` stores the CONditional DENSity propagATION tracker state. The information about the algorithm can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/ISARD1/condensation.html.

---

cvCreateConDensation

Allocates the ConDensation filter structure.

```
CvConDensation* cvCreateConDensation(
           int dynam_params,
           int measure_params,
           int sample_count );
```

dynam_params Dimension of the state vector

measure_params Dimension of the measurement vector

sample_count Number of samples

The function creates a `CvConDensation` structure and returns a pointer to the structure.

---

cvConDensInitSampleSet

Initializes the sample set for the ConDensation algorithm.

```
void cvConDensInitSampleSet( CvConDensation* condens,
           CvMat* lower_bound,
           CvMat* upper_bound );
```

condens Pointer to a structure to be initialized

lower_bound Vector of the lower boundary for each dimension

upper_bound Vector of the upper boundary for each dimension

The function fills the samples arrays in the structure `condens` with values within the specified ranges.

CvKalman

Kalman filter state.

```
typedef struct CvKalman
{
    int MP;                     /* number of measurement vector dimensions */
    int DP;                     /* number of state vector dimensions */
```

```
    int CP;                     /* number of control vector dimensions */

    /* backward compatibility fields */
#if 1
    float* PosterState;         /* =state_pre->data.fl */
    float* PriorState;          /* =state_post->data.fl */
    float* DynamMatr;           /* =transition_matrix->data.fl */
    float* MeasurementMatr;     /* =measurement_matrix->data.fl */
    float* MNCovariance;        /* =measurement_noise_cov->data.fl */
    float* PNCovariance;        /* =process_noise_cov->data.fl */
    float* KalmGainMatr;        /* =gain->data.fl */
    float* PriorErrorCovariance;/* =error_cov_pre->data.fl */
    float* PosterErrorCovariance;/* =error_cov_post->data.fl */
    float* Temp1;               /* temp1->data.fl */
    float* Temp2;               /* temp2->data.fl */
#endif

    CvMat* state_pre;           /* predicted state (x'(k)):
                                   x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* corrected state (x(k)):
                                   x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* state transition matrix (A) */
    CvMat* control_matrix;      /* control matrix (B)
                                   (it is not used if there is no control)*/
    CvMat* measurement_matrix;  /* measurement matrix (H) */
    CvMat* process_noise_cov;   /* process noise covariance matrix (Q) */
    CvMat* measurement_noise_cov; /* measurement noise covariance matrix (R) */
    CvMat* error_cov_pre;       /* priori error estimate covariance matrix (P'(k)):
                                   P'(k)=A*P(k-1)*At + Q*/
    CvMat* gain;                /* Kalman gain matrix (K(k)):
                                   K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* posteriori error estimate covariance matrix (P(k)):
                                   P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

The structure `CvKalman` is used to keep the Kalman filter state. It is created by the cvCreateKalman function, updated by the cvKalmanPredict and cvKalmanCorrect functions and released by the cvReleaseKalman function. Normally, the structure is used for the standard Kalman filter (notation and the formulas below are borrowed from the excellent Kalman tutorial [**?**])

$$x_k = A \cdot x_{k-1} + B \cdot u_k + w_k$$
$$z_k = H \cdot x_k + v_k$$

where:

x_k (x_{k-1})	state of the system at the moment *k* (*k-1*)
z_k	measurement of the system state at the moment *k*
u_k	external control applied at the moment *k*

w_k and v_k are normally-distributed process and measurement noise, respectively:

$$p(w) \sim N(0, Q)$$
$$p(v) \sim N(0, R)$$

that is,
Q process noise covariance matrix, constant or variable,
R measurement noise covariance matrix, constant or variable

In the case of the standard Kalman filter, all of the matrices: A, B, H, Q and R are initialized once after the cvCvKalman structure is allocated via cvCreateKalman. However, the same structure and the same functions may be used to simulate the extended Kalman filter by linearizing the extended Kalman filter equation in the current system state neighborhood, in this case A, B, H (and, probably, Q and R) should be updated on every step.

---

cvCreateKalman

Allocates the Kalman filter structure.

```
CvKalman* cvCreateKalman(
            int dynam_params,
            int measure_params,
            int control_params=0 );
```

dynam_params dimensionality of the state vector

measure_params dimensionality of the measurement vector

control_params dimensionality of the control vector

The function allocates cvCvKalman and all its matrices and initializes them somehow.

cvKalmanCorrect

Adjusts the model state.

```
const CvMat* cvKalmanCorrect( CvKalman* kalman, const CvMat*
measurement );
```

kalman Pointer to the structure to be updated

measurement CvMat containing the measurement vector

```
#define cvKalmanUpdateByMeasurement cvKalmanCorrect
```

The function adjusts the stochastic model state on the basis of the given measurement of the model state:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

where

z_k	given measurement (`mesurement` parameter)
K_k	Kalman "gain" matrix.

The function stores the adjusted state at `kalman->state_post` and returns it on output.

Example. Using Kalman filter to track a rotating point

```
#include "cv.h"
#include "highgui.h"
#include <math.h>

int main(int argc, char** argv)
{
    /* A matrix data */
    const float A[] = { 1, 1, 0, 1 };

    IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
    CvKalman* kalman = cvCreateKalman( 2, 1, 0 );
    /* state is (phi, delta_phi) - angle and angle increment */
    CvMat* state = cvCreateMat( 2, 1, CV_32FC1 );
    CvMat* process_noise = cvCreateMat( 2, 1, CV_32FC1 );
    /* only phi (angle) is measured */
    CvMat* measurement = cvCreateMat( 1, 1, CV_32FC1 );
```

```
CvRandState rng;
int code = -1;

cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

cvZero( measurement );
cvNamedWindow( "Kalman", 1 );

for(;;)
{
    cvRandSetRange( &rng, 0, 0.1, 0 );
    rng.disttype = CV_RAND_NORMAL;

    cvRand( &rng, state );

    memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
    cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1) );
    cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-5) );
    cvSetIdentity( kalman->measurement_noise_cov, cvRealScalar(1e-1) );
    cvSetIdentity( kalman->error_cov_post, cvRealScalar(1));
    /* choose random initial state */
    cvRand( &rng, kalman->state_post );

    rng.disttype = CV_RAND_NORMAL;

    for(;;)
    {
        #define calc_point(angle)                                      \
            cvPoint( cvRound(img->width/2 + img->width/3*cos(angle)),  \
                     cvRound(img->height/2 - img->width/3*sin(angle)))

        float state_angle = state->data.fl[0];
        CvPoint state_pt = calc_point(state_angle);

        /* predict point position */
        const CvMat* prediction = cvKalmanPredict( kalman, 0 );
        float predict_angle = prediction->data.fl[0];
        CvPoint predict_pt = calc_point(predict_angle);
        float measurement_angle;
        CvPoint measurement_pt;

        cvRandSetRange( &rng,
                        0,
                        sqrt(kalman->measurement_noise_cov->data.fl[0]),
                        0 );
```

```
            cvRand( &rng, measurement );

            /* generate measurement */
            cvMatMulAdd( kalman->measurement_matrix, state, measurement, measurement );

            measurement_angle = measurement->data.fl[0];
            measurement_pt = calc_point(measurement_angle);

            /* plot points */
            #define draw_cross( center, color, d )                          \
                cvLine( img, cvPoint( center.x - d, center.y - d ),          \
                             cvPoint( center.x + d, center.y + d ),          \
                             color, 1, 0 );                                  \
                cvLine( img, cvPoint( center.x + d, center.y - d ),          \
                             cvPoint( center.x - d, center.y + d ),          \
                             color, 1, 0 )

            cvZero( img );
            draw_cross( state_pt, CV_RGB(255,255,255), 3 );
            draw_cross( measurement_pt, CV_RGB(255,0,0), 3 );
            draw_cross( predict_pt, CV_RGB(0,255,0), 3 );
            cvLine( img, state_pt, predict_pt, CV_RGB(255,255,0), 3, 0 );

            /* adjust Kalman filter state */
            cvKalmanCorrect( kalman, measurement );

            cvRandSetRange( &rng,
                            0,
                            sqrt(kalman->process_noise_cov->data.fl[0]),
                            0 );
            cvRand( &rng, process_noise );
            cvMatMulAdd( kalman->transition_matrix,
                         state,
                         process_noise,
                         state );

            cvShowImage( "Kalman", img );
            code = cvWaitKey( 100 );

            if( code > 0 ) /* break current simulation by pressing a key */
                break;
        }
        if( code == 27 ) /* exit by ESCAPE */
            break;
    }
```

```
    return 0;
}
```

---

cvKalmanPredict

Estimates the subsequent model state.

```
const CvMat* cvKalmanPredict(
           CvKalman* kalman,
           const CvMat* control=NULL );
```

```
#define cvKalmanUpdateByTime cvKalmanPredict
```

kalman Kalman filter state

control Control vector u_k, should be NULL iff there is no external control (`control_params` =0)

The function estimates the subsequent stochastic model state by its current state and stores it at `kalman->state_pre`:

$$x'_k = A \cdot x_{k-1} + B \cdot u_k$$
$$P'_k = A \cdot P_{k-1} + A^T + Q$$

where

x'_k	is predicted state `kalman->state_pre`,
x_{k-1}	is corrected state on the previous step `kalman->state_post` (should be initialized somehow in the beginning, zero vector by default),
u_k	is external control (`control` parameter),
P'_k	is priori error covariance matrix `kalman->error_cov_pre`
P_{k-1}	is posteriori error covariance matrix on the previous step `kalman->error_cov_post` (should be initialized somehow in the beginning, identity matrix by default),

The function returns the estimated state.

cvMeanShift

Finds the object center on back projection.

```
int cvMeanShift(
           const CvArr* prob_image,
           CvRect window,
           CvTermCriteria criteria,
           CvConnectedComp* comp );
```

prob_image Back projection of the object histogram (see cvCalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

The function iterates to find the object center given its back projection and initial position of search window. The iterations are made until the search window center moves by less than the given value and/or until the function has done the maximum number of iterations. The function returns the number of iterations made.

cvMultiplyAcc

Adds the product of two input images to the accumulator.

```
void cvMultiplyAcc(
           const CvArr* image1,
           const CvArr* image2,
           CvArr* acc,
           const CvArr* mask=NULL );
```

image1 First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

image2 Second input image, the same format as the first one

acc Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `acc`:

$$\texttt{acc}(x,y) \leftarrow \texttt{acc}(x,y) + \texttt{image1}(x,y) \cdot \texttt{image2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cvReleaseConDensation

Deallocates the ConDensation filter structure.

```
void cvReleaseConDensation( CvConDensation** condens );
```

condens Pointer to the pointer to the structure to be released

The function releases the structure `condens`) and frees all memory previously allocated for the structure.

---

cvReleaseKalman

Deallocates the Kalman filter structure.

```
void cvReleaseKalman(
           CvKalman** kalman );
```

kalman double pointer to the Kalman filter structure

The function releases the structure cvCvKalman and all of the underlying matrices.

---

cvRunningAvg

Updates the running average.

```
void cvRunningAvg(
           const CvArr* image,
           CvArr* acc,
           double alpha,
           const CvArr* mask=NULL );
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

acc Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

alpha Weight of input image

mask Optional operation mask

The function calculates the weighted sum of the input image `image` and the accumulator `acc` so that `acc` becomes a running average of frame sequence:

$$\texttt{acc}(x,y) \leftarrow (1-\alpha) \cdot \texttt{acc}(x,y) + \alpha \cdot \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

where α regulates the update speed (how fast the accumulator forgets about previous frames).

---

cvSegmentMotion

Segments a whole motion into separate moving parts.

```
CvSeq* cvSegmentMotion(
           const CvArr* mhi,
           CvArr* seg_mask,
           CvMemStorage* storage,
           double timestamp,
           double seg_thresh );
```

mhi Motion history image

seg_mask Image where the mask found should be stored, single-channel, 32-bit floating-point

storage Memory storage that will contain a sequence of motion connected components

timestamp Current time in milliseconds or other units

seg_thresh Segmentation threshold; recommended to be equal to the interval between motion history "steps" or greater

The function finds all of the motion segments and marks them in `seg_mask` with individual values (1,2,...). It also returns a sequence of cvCvConnectedComp structures, one for each motion component. After that the motion direction for every component can be calculated with cvCalcGlobalOrientation using the extracted mask of the particular component cvCmp.

---

cvSnakeImage

Changes the contour position to minimize its energy.

```
void cvSnakeImage(
           const IplImage* image,
           CvPoint* points,
           int length,
           float* alpha,
           float* beta,
           float* gamma,
           int coeff_usage,
           CvSize win,
           CvTermCriteria criteria,
           int calc_gradient=1 );
```

image The source image or external energy field

points Contour points (snake)

length Number of points in the contour

alpha Weight[s] of continuity energy, single float or array of `length` floats, one for each contour point

beta Weight[s] of curvature energy, similar to `alpha`

gamma Weight[s] of image energy, similar to `alpha`

coeff_usage Different uses of the previous three parameters:

CV_VALUE indicates that each of `alpha, beta, gamma` is a pointer to a single value to be used for all points;

CV_ARRAY indicates that each of `alpha, beta, gamma` is a pointer to an array of coefficients different for all the points of the snake. All the arrays must have the size equal to the contour size.

win Size of neighborhood of every point used to search the minimum, both `win.width` and `win.height` must be odd

criteria Termination criteria

calc_gradient Gradient flag; if not 0, the function calculates the gradient magnitude for every image pixel and consideres it as the energy field, otherwise the input image itself is considered

The function updates the snake in order to minimize its total energy that is a sum of internal energy that depends on the contour shape (the smoother contour is, the smaller internal energy is) and external energy that depends on the energy field and reaches minimum at the local energy extremums that correspond to the image edges in the case of using an image gradient.

The parameter `criteria.epsilon` is used to define the minimal number of points that must be moved during any iteration to keep the iteration process running.

If at some iteration the number of moved points is less than `criteria.epsilon` or the function performed `criteria.max_iter` iterations, the function terminates.

---

cvSquareAcc

Adds the square of the source image to the accumulator.

```
void cvSquareAcc(
           const CvArr* image,
           CvArr* sqsum,
           const CvArr* mask=NULL );
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

sqsum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum`:

$$\texttt{sqsum}(x,y) \leftarrow \texttt{sqsum}(x,y) + \texttt{image}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cvUpdateMotionHistory

Updates the motion history image by a moving silhouette.

```
void cvUpdateMotionHistory(
           const CvArr* silhouette,
           CvArr* mhi,
           double timestamp,
           double duration );
```

silhouette Silhouette mask that has non-zero pixels where the motion occurs

mhi Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp Current time in milliseconds or other units

duration Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

2.7 Structural Analysis and Shape Descriptors

---

cvApproxChains

Approximates Freeman chain(s) with a polygonal curve.

```
CvSeq* cvApproxChains(
           CvSeq* src_seq,
           CvMemStorage* storage,
           int method=CV_CHAIN_APPROX_SIMPLE,
           double parameter=0,
           int minimal_perimeter=0,
           int recursive=0 );
```

src_seq Pointer to the chain that can refer to other chains

storage Storage location for the resulting polylines

method Approximation method (see the description of the function cvFindContours)

parameter Method parameter (not used now)

minimal_perimeter Approximates only those contours whose perimeters are not less than `minimal_perimeter`. Other chains are removed from the resulting structure

recursive If not 0, the function approximates all chains that access can be obtained to from `src_seq` by using the `h_next` or `v_next links`. If 0, the single chain is approximated

This is a stand-alone approximation routine. The function `cvApproxChains` works exactly in the same way as cvFindContours with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via the `v_next` or `h_next` fields of the returned structure.

---

cvApproxPoly

Approximates polygonal curve(s) with the specified precision.

```
CvSeq* cvApproxPoly(
           const void* src_seq,
           int header_size,
           CvMemStorage* storage,
           int method,
           double parameter,
           int parameter2=0 );
```

src_seq Sequence of an array of points

header_size Header size of the approximated curve[s]

storage Container for the approximated contours. If it is NULL, the input sequences' storage is used

method Approximation method; only `CV_POLY_APPROX_DP` is supported, that corresponds to the Douglas-Peucker algorithm

parameter Method-specific parameter; in the case of `CV_POLY_APPROX_DP` it is a desired approximation accuracy

parameter2 If case if `src_seq` is a sequence, the parameter determines whether the single sequence should be approximated or all sequences on the same level or below `src_seq` (see cvFindContours for description of hierarchical contour structures). If `src_seq` is an array CvMat* of points, the parameter specifies whether the curve is closed (`parameter2`!=0) or not (`parameter2` =0)

The function approximates one or more curves and returns the approximation result[s]. In the case of multiple curves, the resultant tree will have the same structure as the input one (1:1 correspondence).

---

cvArcLength

Calculates the contour perimeter or the curve length.

```
double cvArcLength(
           const void* curve,
           CvSlice slice=CV_WHOLE_SEQ,
           int isClosed=-1 );
```

curve Sequence or array of the curve points

slice Starting and ending points of the curve, by default, the whole curve length is calculated

isClosed Indicates whether the curve is closed or not. There are 3 cases:

- `isClosed` $= 0$ the curve is assumed to be unclosed.
- `isClosed` > 0 the curve is assumed to be closed.

- `isClosed` < 0 if curve is sequence, the flag `CV_SEQ_FLAG_CLOSED` of `((CvSeq*)curve)->flags` is checked to determine if the curve is closed or not, otherwise (curve is represented by array (CvMat*) of points) it is assumed to be unclosed.

The function calculates the length or curve as the sum of lengths of segments between subsequent points

---

cvBoundingRect

Calculates the up-right bounding rectangle of a point set.

```
CvRect cvBoundingRect( CvArr* points, int update=0 );
```

points 2D point set, either a sequence or vector (`CvMat`) of points

update The update flag. See below.

The function returns the up-right bounding rectangle for a 2d point set. Here is the list of possible combination of the flag values and type of `points`:

update	points	action
0	`CvContour`	the bounding rectangle is not calculated, but it is taken from `rect` field of the contour header.
1	`CvContour`	the bounding rectangle is calculated and written to `rect` field of the contour header.
0	`CvSeq` or `CvMat`	the bounding rectangle is calculated and returned.
1	`CvSeq` or `CvMat`	runtime error is raised.

---

cvBoxPoints

Finds the box vertices.

```
void cvBoxPoints(
           CvBox2D box,
           CvPoint2D32f pt[4] );
```

box Box

points Array of vertices

The function calculates the vertices of the input 2d box. Here is the function code:

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    float a = (float)cos(box.angle)*0.5f;
    float b = (float)sin(box.angle)*0.5f;

    pt[0].x = box.center.x - a*box.size.height - b*box.size.width;
    pt[0].y = box.center.y + b*box.size.height - a*box.size.width;
    pt[1].x = box.center.x + a*box.size.height - b*box.size.width;
    pt[1].y = box.center.y - b*box.size.height - a*box.size.width;
    pt[2].x = 2*box.center.x - pt[0].x;
    pt[2].y = 2*box.center.y - pt[0].y;
    pt[3].x = 2*box.center.x - pt[1].x;
    pt[3].y = 2*box.center.y - pt[1].y;
}
```

---

cvCalcPGH

Calculates a pair-wise geometrical histogram for a contour.

```
void cvCalcPGH( const CvSeq* contour, CvHistogram* hist );
```

contour Input contour. Currently, only integer point coordinates are allowed

hist Calculated histogram; must be two-dimensional

The function calculates a 2D pair-wise geometrical histogram (PGH), described in cvlivarinen97 for the contour. The algorithm considers every pair of contour edges. The angle between the edges and the minimum/maximum distances are determined for every pair. To do this each of the edges in turn is taken as the base, while the function loops through all the other edges. When the base edge and any other edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented (that is, the histogram is transposed relatively to the cvlivarninen97 definition). The histogram can be used for contour matching.

cvCalcEMD2

Computes the ”minimal work” distance between two weighted point configurations.

```
float cvCalcEMD2(
            const CvArr* signature1,
            const CvArr* signature2,
            int distance_type,
            CvDistanceFunction distance_func=NULL,
            const CvArr* cost_matrix=NULL,
            CvArr* flow=NULL,
            float* lower_bound=NULL,
            void* userdata=NULL );
```

signature1 First signature, a $\texttt{size1} \times \texttt{dims} + 1$ floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used

signature2 Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different, in this case an extra ”dummy” point is added to either `signature1` or `signature2`

distance_type Metrics used; `CV_DIST_L1, CV_DIST_L2`, and `CV_DIST_C` stand for one of the standard metrics; `CV_DIST_USER` means that a user-defined function `distance_func` or pre-calculated `cost_matrix` is used

distance_func The user-defined distance function. It takes coordinates of two points and returns the distance between the points

cost_matrix The user-defined $\texttt{size1} \times \texttt{size2}$ cost matrix. At least one of `cost_matrix` and `distance_func` must be NULL. Also, if a cost matrix is used, lower boundary (see below) can not be calculated, because it needs a metric function

flow The resultant $\texttt{size1} \times \texttt{size2}$ flow matrix: $\texttt{flow}_{i,j}$ is a flow from i th point of `signature1` to j th point of `signature2`

lower_bound Optional input/output parameter: lower boundary of distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or if the signatures consist of weights only (i.e. the signature matrices have a single column).

The user **must** initialize `*lower_bound`. If the calculated distance between mass centers is greater or equal to `*lower_bound` (it means that the signatures are far enough) the function does not calculate EMD. In any case `*lower_bound` is set to the calculated distance between mass centers on return. Thus, if user wants to calculate both distance between mass centers and EMD, `*lower_bound` should be set to 0

userdata Pointer to optional data that is passed into the user-defined distance function

```
typedef float (*CvDistanceFunction)(const float* f1, const float* f2, void* userdata);
```

The function computes the earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the applications described in cvRubnerSept98 is multi-dimensional histogram comparison for image retrieval. EMD is a a transportation problem that is solved using some modification of a simplex algorithm, thus the complexity is exponential in the worst case, though, on average it is much faster. In the case of a real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

---

cvCheckContourConvexity

Tests contour convexity.

```
int cvCheckContourConvexity( const CvArr* contour );
```

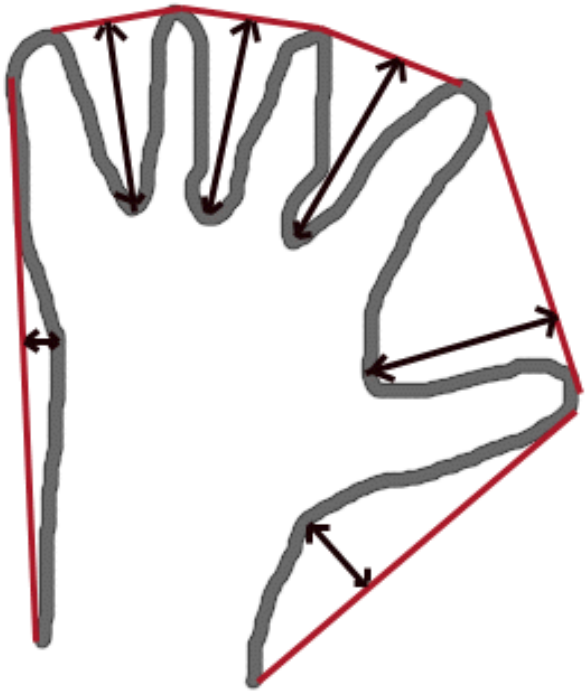
contour Tested contour (sequence or array of points)

The function tests whether the input contour is convex or not. The contour must be simple, without self-intersections.

CvConvexityDefect

Structure describing a single contour convexity defect.

```
typedef struct CvConvexityDefect
{
    CvPoint* start; /* point of the contour where the defect begins */
    CvPoint* end; /* point of the contour where the defect ends */
    CvPoint* depth_point; /* the farthest from the convex hull point within the defect */
    float depth; /* distance between the farthest point and the convex hull */
} CvConvexityDefect;
```



---

cvContourArea

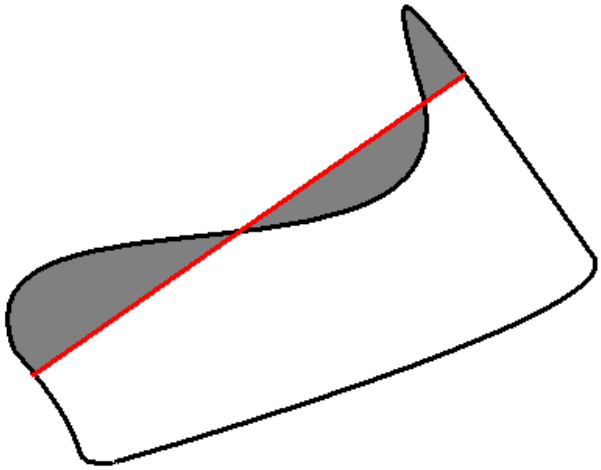
Calculates the area of a whole contour or a contour section.

```
double cvContourArea(
           const CvArr* contour,
           CvSlice slice=CV_WHOLE_SEQ );
```

contour Contour (sequence or array of vertices)

slice Starting and ending points of the contour section of interest, by default, the area of the whole contour is calculated

The function calculates the area of a whole contour or a contour section. In the latter case the total area bounded by the contour arc and the chord connecting the 2 selected points is calculated as shown on the picture below:



Orientation of the contour affects the area sign, thus the function may return a *negative* result. Use the `fabs()` function from C runtime to get the absolute value of the area.

---

cvContourFromContourTree

Restores a contour from the tree.

```
CvSeq* cvContourFromContourTree(
           const CvContourTree* tree,
           CvMemStorage* storage,
           CvTermCriteria criteria );
```

`tree` Contour tree

`storage` Container for the reconstructed contour

`criteria` Criteria, where to stop reconstruction

The function restores the contour from its binary tree representation. The parameter `criteria` determines the accuracy and/or the number of tree levels used for reconstruction, so it is possible to build an approximated contour. The function returns the reconstructed contour.

---

cvConvexHull2

Finds the convex hull of a point set.

```
CvSeq* cvConvexHull2(
           const CvArr* input,
           void* storage=NULL,
           int orientation=CV_CLOCKWISE,
           int return_points=0 );
```

points Sequence or array of 2D points with 32-bit integer or floating-point coordinates

storage The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is an array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified as to truncate the array down to the hull size. If `storage` is NULL then the convex hull will be stored in the same storage as the input sequence

orientation Desired orientation of convex hull: `CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`

return_points If non-zero, the points themselves will be stored in the hull instead of indices if `storage` is an array, or pointers if `storage` is memory storage

The function finds the convex hull of a 2D point set using Sklansky's algorithm. If `storage` is memory storage, the function creates a sequence containing the hull points or pointers to them, depending on `return_points` value and returns the sequence on output. If `storage` is a CvMat, the function returns NULL.

```
#include "cv.h"
#include "highgui.h"
#include <stdlib.h>

#define ARRAY  0 /* switch between array/sequence method by replacing 0<=>1 */

void main( int argc, char** argv )
{
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    cvNamedWindow( "hull", 1 );

#if !ARRAY
        CvMemStorage* storage = cvCreateMemStorage();
#endif

    for(;;)
    {
```

```
        int i, count = rand()%100 + 1, hullcount;
        CvPoint pt0;
#if !ARRAY
        CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2,
                                    sizeof(CvContour),
                                    sizeof(CvPoint),
                                    storage );
        CvSeq* hull;

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            cvSeqPush( ptseq, &pt0 );
        }
        hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
        hullcount = hull->total;
#else
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
        int* hull = (int*)malloc( count * sizeof(hull[0]));
        CvMat point_mat = cvMat( 1, count, CV_32SC2, points );
        CvMat hull_mat = cvMat( 1, count, CV_32SC1, hull );

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            points[i] = pt0;
        }
        cvConvexHull2( &point_mat, &hull_mat, CV_CLOCKWISE, 0 );
        hullcount = hull_mat.cols;
#endif
        cvZero( img );
        for( i = 0; i < count; i++ )
        {
#if !ARRAY
            pt0 = *CV_GET_SEQ_ELEM( CvPoint, ptseq, i );
#else
            pt0 = points[i];
#endif
            cvCircle( img, pt0, 2, CV_RGB( 255, 0, 0 ), CV_FILLED );
        }

#if !ARRAY
        pt0 = **CV_GET_SEQ_ELEM( CvPoint*, hull, hullcount - 1 );
```

```
#else
        pt0 = points[hull[hullcount-1]];
#endif

        for( i = 0; i < hullcount; i++ )
        {
#if !ARRAY
            CvPoint pt = **CV_GET_SEQ_ELEM( CvPoint*, hull, i );
#else
            CvPoint pt = points[hull[i]];
#endif
            cvLine( img, pt0, pt, CV_RGB( 0, 255, 0 ));
            pt0 = pt;
        }

        cvShowImage( "hull", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;

#if !ARRAY
        cvClearMemStorage( storage );
#else
        free( points );
        free( hull );
#endif
    }
}
```

---

cvConvexityDefects

Finds the convexity defects of a contour.

```
CvSeq* cvConvexityDefects(
            const CvArr* contour,
            const CvArr* convexhull,
            CvMemStorage* storage=NULL );
```

contour Input contour

convexhull Convex hull obtained using cvConvexHull2 that should contain pointers or indices to the contour points, not the hull points themselves (the `return_points` parameter in cvConvexHull2 should be 0)

storage Container for the output sequence of convexity defects. If it is NULL, the contour or hull (in that order) storage is used

The function finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect structures.

---

cvCreateContourTree

Creates a hierarchical representation of a contour.

```
CvContourTree* cvCreateContourTree(
            const CvSeq* contour,
            CvMemStorage* storage,
            double threshold );
```

contour Input contour

storage Container for output tree

threshold Approximation accuracy

The function creates a binary tree representation for the input `contour` and returns the pointer to its root. If the parameter `threshold` is less than or equal to 0, the function creates a full binary tree representation. If the threshold is greater than 0, the function creates a representation with the precision `threshold`: if the vertices with the interceptive area of its base line are less than `threshold`, the tree should not be built any further. The function returns the created tree.

---

cvEndFindContours

Finishes the scanning process.

```
CvSeq* cvEndFindContours(
            CvContourScanner* scanner );
```

scanner Pointer to the contour scanner

The function finishes the scanning process and returns a pointer to the first contour on the highest level.

---

cvFindContours

Finds the contours in a binary image.

```
int cvFindContours(
            CvArr* image,
            CvMemStorage* storage,
            CvSeq** first_contour,
            int header_size=sizeof(CvContour),
            int mode=CV_RETR_LIST,
            int method=CV_CHAIN_APPROX_SIMPLE,
            CvPoint offset=cvPoint(0,0) );
```

image The source, an 8-bit single channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary`. To get such a binary image from grayscale, one may use cvThreshold, cvAdaptiveThreshold or cvCanny. The function modifies the source image's content

storage Container of the retrieved contours

first_contour Output parameter, will contain the pointer to the first outer contour

header_size Size of the sequence header, $\geq$ `sizeof(CvChain)` if `method` $=$ `CV_CHAIN_CODE`, and $\geq$ `sizeof(CvContour)` otherwise

mode Retrieval mode

- **CV_RETR_EXTERNAL** retrives only the extreme outer contours
- **CV_RETR_LIST** retrieves all of the contours and puts them in the list
- **CV_RETR_CCOMP** retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes
- **CV_RETR_TREE** retrieves all of the contours and reconstructs the full hierarchy of nested contours

method Approximation method (for all the modes, except CV_LINK_RUNS, which uses built-in approximation)

- **CV_CHAIN_CODE** outputs contours in the Freeman chain code. All other methods output polygons (sequences of vertices)
- **CV_CHAIN_APPROX_NONE** translates all of the points from the chain code into points
- **CV_CHAIN_APPROX_SIMPLE** compresses horizontal, vertical, and diagonal segments and leaves only their end points
- **CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS** applies one of the flavors of the Teh-Chin chain approximation algorithm.
- **CV_LINK_RUNS** uses a completely different contour retrieval algorithm by linking horizontal segments of 1's. Only the CV_RETR_LIST retrieval mode can be used with this method.

offset Offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context

The function retrieves contours from the binary image and returns the number of retrieved contours. The pointer first_contour is filled by the function. It will contain a pointer to the first outermost contour or NULL if no contours are detected (if the image is completely black). Other contours may be reached from first_contour using the h_next and v_next links. The sample in the cvDrawContours discussion shows how to use contours for connected component detection. Contours can be also used for shape analysis and object recognition - see squares.c in the OpenCV sample directory.

---

cvFindNextContour

Finds the next contour in the image.

```
CvSeq* cvFindNextContour(
            CvContourScanner scanner );
```

scanner Contour scanner initialized by cvStartFindContours

The function locates and retrieves the next contour in the image and returns a pointer to it. The function returns NULL if there are no more contours.

cvFitEllipse2

Fits an ellipse around a set of 2D points.

```
CvBox2D cvFitEllipse2(
           const CvArr* points );
```

points Sequence or array of points

The function calculates the ellipse that fits best (in least-squares sense) around a set of 2D points. The meaning of the returned structure fields is similar to those in cvEllipse except that `size` stores the full lengths of the ellipse axises, not half-lengths.

cvFitLine

Fits a line to a 2D or 3D point set.

```
void cvFitLine(
           const CvArr* points,
           int dist_type,
           double param,
           double reps,
           double aeps,
           float* line );
```

points Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates

dist_type The distance used for fitting (see the discussion)

param Numerical parameter (`C`) for some types of distances, if 0 then some optimal value is chosen

reps Sufficient accuracy for the radius (distance between the coordinate origin and the line). 0.01 is a good default value.

aeps Sufficient accuracy for the angle. 0.01 is a good default value.

line The output line parameters. In the case of a 2d fitting, it is an array of 4 floats `(vx, vy, x0, y0)` where `(vx, vy)` is a normalized vector collinear to the line and `(x0, y0)` is some point on the line. in the case of a 3D fitting it is an array of 6 floats `(vx, vy, vz, x0, y0, z0)` where `(vx, vy, vz)` is a normalized vector collinear to the line and `(x0, y0, z0)` is some point on the line

The function fits a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i th point and the line and $\rho(r)$ is a distance function, one of:

dist_type=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

dist_type=CV_DIST_L1

$$\rho(r) = r$$

dist_type=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

dist_type=CV_DIST_FAIR

$$\rho\left(r\right) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right) \quad \text{where} \quad C = 1.3998$$

dist_type=CV_DIST_WELSCH

$$\rho\left(r\right) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

dist_type=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

---

cvGetCentralMoment

Retrieves the central moment from the moment state structure.

```
double cvGetCentralMoment(
           CvMoments* moments,
           int x_order,
           int y_order );
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, x_order $>= 0$

y_order y order of the retrieved moment, y_order $>= 0$ and x_order + y_order $<= 3$

The function retrieves the central moment, which in the case of image moments is defined as:

$$\mu_{x\_order,\,y\_order} = \sum_{x,y}(I(x,y)\cdot(x-x_c)^{x\_order}\cdot(y-y_c)^{y\_order})$$

where x_c, y_c are the coordinates of the gravity center:

$$x_c = \frac{M_{10}}{M_{00}}, y_c = \frac{M_{01}}{M_{00}}$$

---

cvGetHuMoments

Calculates the seven Hu invariants.

```
void cvGetHuMoments( const CvMoments* moments,CvHuMoments* hu );
```

moments The input moments, computed with cvMoments

hu The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment, that are defined as:

$$\begin{aligned}
hu_1 &= \eta_{20} + \eta_{02}\\
hu_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2\\
hu_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2\\
hu_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2\\
hu_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]\\
hu_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})\\
hu_7 &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}$$

where η_{ji} denote the normalized central moments.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

cvGetNormalizedCentralMoment

Retrieves the normalized central moment from the moment state structure.

```
double cvGetNormalizedCentralMoment(
           CvMoments* moments,
           int x_order,
           int y_order );
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, $\texttt{x\_order} >= 0$

y_order y order of the retrieved moment, $\texttt{y\_order} >= 0$ and $\texttt{x\_order} + \texttt{y\_order} <= 3$

The function retrieves the normalized central moment:

$$\eta_{x\_order, y\_order} = \frac{\mu_{x\_order, y\_order}}{M_{00}^{(y\_order+x\_order)/2+1}}$$

cvGetSpatialMoment

Retrieves the spatial moment from the moment state structure.

```
double cvGetSpatialMoment(
           CvMoments* moments,
           int x_order,
           int y_order );
```

moments The moment state, calculated by cvMoments

x_order x order of the retrieved moment, $\texttt{x\_order} >= 0$

y_order y order of the retrieved moment, $\texttt{y\_order} >= 0$ and $\texttt{x\_order} + \texttt{y\_order} <= 3$

The function retrieves the spatial moment, which in the case of image moments is defined as:

$$M_{x\_order, y\_order} = \sum_{x,y} (I(x,y) \cdot x^{x\_order} \cdot y^{y\_order})$$

where $I(x,y)$ is the intensity of the pixel (x,y).

cvMatchContourTrees

Compares two contours using their tree representations.

```
double cvMatchContourTrees(
           const CvContourTree* tree1,
           const CvContourTree* tree2,
           int method,
           double threshold );
```

tree1 First contour tree

tree2 Second contour tree

method Similarity measure, only CV_CONTOUR_TREES_MATCH_I1 is supported

threshold Similarity threshold

The function calculates the value of the matching measure for two contour trees. The similarity measure is calculated level by level from the binary tree roots. If at a certain level the difference between contours becomes less than threshold, the reconstruction process is interrupted and the current difference is returned.

cvMatchShapes

Compares two shapes.

```
double cvMatchShapes(
           const void* object1,
           const void* object2,
           int method,
           double parameter=0 );
```

object1 First contour or grayscale image

object2 Second contour or grayscale image

method Comparison method; CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3

parameter Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu moments (see cvGetHuMoments) (A is `object1`, B is `object2`):

method=CV_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1...7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1...7} \left| m_i^A - m_i^B \right|$$

method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1...7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

where

$$m_i^A = sign(h_i^A) \cdot \log h_i^A m_i^B = sign(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

---

cvMinAreaRect2

Finds the circumscribed rectangle of minimal area for a given 2D point set.

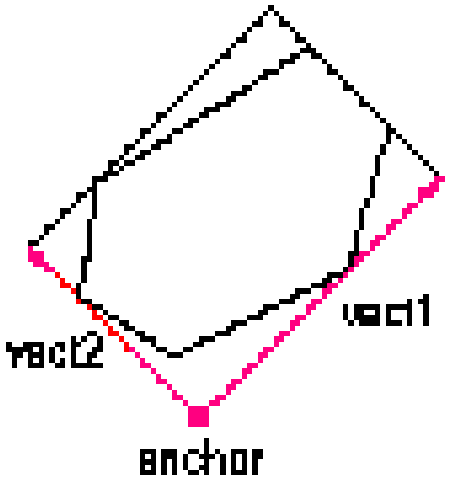
```
CvBox2D cvMinAreaRect2(
           const CvArr* points,
           CvMemStorage* storage=NULL );
```

points Sequence or array of points

storage Optional temporary memory storage

The function finds a circumscribed rectangle of the minimal area for a 2D point set by building a convex hull for the set and applying the rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour



---

cvMinEnclosingCircle

Finds the circumscribed circle of minimal area for a given 2D point set.

```
int cvMinEnclosingCircle(
           const CvArr* points,
           CvPoint2D32f* center,
           float* radius );
```

points Sequence or array of 2D points

center Output parameter; the center of the enclosing circle

radius Output parameter; the radius of the enclosing circle

The function finds the minimal circumscribed circle for a 2D point set using an iterative algorithm. It returns nonzero if the resultant circle contains all the input points and zero otherwise (i.e. the algorithm failed).

cvMoments

Calculates all of the moments up to the third order of a polygon or rasterized shape.

```
void cvMoments(
           const CvArr* arr,
           CvMoments* moments,
           int binary=0 );
```

arr Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points or a vector of points)

moments Pointer to returned moment's state structure

binary (For images only) If the flag is non-zero, all of the zero pixel values are treated as zeroes, and all of the others are treated as 1's

The function calculates spatial and central moments up to the third order and writes them to `moments`. The moments may then be used then to calculate the gravity center of the shape, its area, main axises and various shape characeteristics including 7 Hu invariants.

cvPointPolygonTest

Point in contour test.

```
double cvPointPolygonTest(
           const CvArr* contour,
           CvPoint2D32f pt,
           int measure_dist );
```

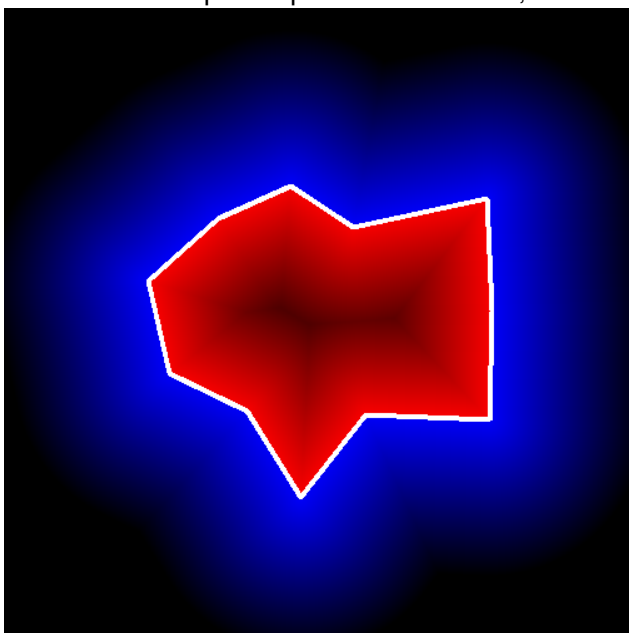
contour Input contour

pt The point tested against the contour

measure_dist If it is non-zero, the function estimates the distance from the point to the nearest contour edge

The function determines whether the point is inside a contour, outside, or lies on an edge (or coinsides with a vertex). It returns positive, negative or zero value, correspondingly. When measure_dist $= 0$, the return value is +1, -1 and 0, respectively. When measure_dist $\neq 0$, it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



---

cvPointSeqFromMat

Initializes a point sequence header from a point vector.

```
CvSeq* cvPointSeqFromMat(
           int seq_kind,
           const CvArr* mat,
           CvContour* contour_header,
           CvSeqBlock* block );
```

seq_kind Type of the point sequence: point set (0), a curve (CV_SEQ_KIND_CURVE), closed curve (CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED) etc.

mat Input matrix. It should be a continuous, 1-dimensional vector of points, that is, it should have type CV_32SC2 or CV_32FC2

contour_header Contour header, initialized by the function

block Sequence block header, initialized by the function

The function initializes a sequence header to create a "virtual" sequence in which elements reside in the specified matrix. No data is copied. The initialized sequence header may be passed to any function that takes a point sequence on input. No extra elements can be added to the sequence, but some may be removed. The function is a specialized variant of cvMakeSeqHeaderForArray and uses the latter internally. It returns a pointer to the initialized contour header. Note that the bounding rectangle (field `rect` of `CvContour` strucuture) is not initialized by the function. If you need one, use cvBoundingRect.

Here is a simple usage example.

```
CvContour header;
CvSeqBlock block;
CvMat* vector = cvCreateMat( 1, 3, CV_32SC2 );

CV_MAT_ELEM( *vector, CvPoint, 0, 0 ) = cvPoint(100,100);
CV_MAT_ELEM( *vector, CvPoint, 0, 1 ) = cvPoint(100,200);
CV_MAT_ELEM( *vector, CvPoint, 0, 2 ) = cvPoint(200,100);

IplImage* img = cvCreateImage( cvSize(300,300), 8, 3 );
cvZero(img);

cvDrawContours( img,
    cvPointSeqFromMat(CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED,
                      vector,
                      &header,
                      &block),
                CV_RGB(255,0,0),
                CV_RGB(255,0,0),
                0, 3, 8, cvPoint(0,0));
```

---

cvReadChainPoint

Gets the next chain point.

```
CvPoint cvReadChainPoint( CvChainPtReader* reader );
```

reader Chain reader state

The function returns the current chain point and updates the reader position.

cvStartFindContours

Initializes the contour scanning process.

```
CvContourScanner cvStartFindContours(
           CvArr* image,
           CvMemStorage* storage,
           int header_size=sizeof(CvContour),
           int mode=CV_RETR_LIST,
           int method=CV_CHAIN_APPROX_SIMPLE,
           CvPoint offset=cvPoint(0,
           0) );
```

image The 8-bit, single channel, binary source image

storage Container of the retrieved contours

header_size Size of the sequence header, $>= sizeof(CvChain)$ if `method` =CV_CHAIN_CODE, and $>= sizeof(CvContour)$ otherwise

mode Retrieval mode; see cvFindContours

method Approximation method. It has the same meaning in cvFindContours, but `CV_LINK_RUNS` can not be used here

offset ROI offset; see cvFindContours

The function initializes and returns a pointer to the contour scanner. The scanner is used in cvFindNextContour to retrieve the rest of the contours.

cvStartReadChainPoints

Initializes the chain reader.

```
void cvStartReadChainPoints( CvChain* chain, CvChainPtReader* reader );
```

The function initializes a special reader.

cvSubstituteContour

Replaces a retrieved contour.

```
void cvSubstituteContour(
            CvContourScanner scanner,
            CvSeq* new_contour );
```

scanner Contour scanner initialized by cvStartFindContours

new_contour Substituting contour

The function replaces the retrieved contour, that was returned from the preceding call of cvFindNextContour and stored inside the contour scanner state, with the user-specified contour. The contour is inserted into the resulting structure, list, two-level hierarchy, or tree, depending on the retrieval mode. If the parameter `new_contour` is `NULL`, the retrieved contour is not included in the resulting structure, nor are any of its children that might be added to this structure later.

2.8 Planar Subdivisions

CvSubdiv2D

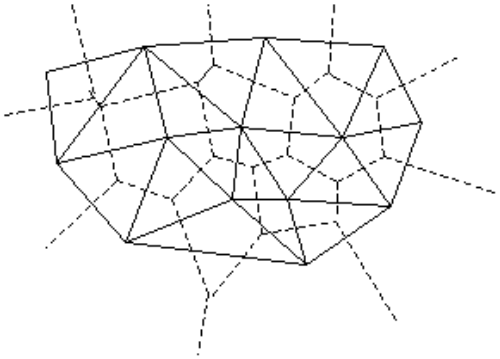
Planar subdivision.

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
CvSubdiv2D;
```

Planar subdivision is the subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on a 2d point set, where the points are linked together and form a planar graph, which, together with a few edges

connecting the exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists a dual subdivision in which facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called a virtual point below) of the dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dotted lines.



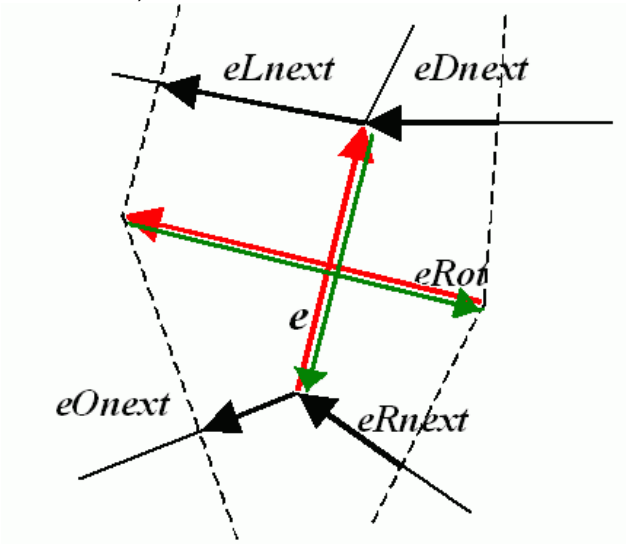
OpenCV subdivides a plane into triangles using Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case the dual subdivision is a Voronoi diagram of the input 2d point set. The subdivisions can be used for the 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) and so forth.

CvQuadEdge2D

Quad-edge of planar subdivision.

```
/* one of edges within quad-edge, lower 2 bits is index (0..3)
   and upper bits are quad-edge pointer */
typedef long CvSubdiv2DEdge;

/* quad-edge structure fields */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```

Quad-edge is a basic element of subdivision containing four edges (e, eRot, reversed e and reversed eRot):



CvSubdiv2DPoint

Point of original or dual subdivision.

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int            flags;      \
    CvSubdiv2DEdge first;      \
    CvPoint2D32f   pt;         \
    int id;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

id This integer can be used to index auxillary data associated with each vertex of the planar subdivision

cvCalcSubdivVoronoi2D

Calculates the coordinates of Voronoi diagram cells.

```
void cvCalcSubdivVoronoi2D(
           CvSubdiv2D* subdiv );
```

subdiv Delaunay subdivision, in which all the points are already added

The function calculates the coordinates of virtual points. All virtual points corresponding to some vertex of the original subdivision form (when connected together) a boundary of the Voronoi cell at that point.

---

cvClearSubdivVoronoi2D

Removes all virtual points.

```
void cvClearSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv Delaunay subdivision

The function removes all of the virtual points. It is called internally in cvCalcSubdivVoronoi2D if the subdivision was modified after previous call to the function.

---

cvCreateSubdivDelaunay2D

Creates an empty Delaunay triangulation.

```
CvSubdiv2D* cvCreateSubdivDelaunay2D(
           CvRect rect,
           CvMemStorage* storage );
```

rect Rectangle that includes all of the 2d points that are to be added to the subdivision

storage Container for subdivision

The function creates an empty Delaunay subdivision, where 2d points can be added using the function cvSubdivDelaunay2DInsert. All of the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

Note that the triangulation is a single large triangle that covers the given rectangle. Hence the three vertices of this triangle are outside the rectangle `rect`.

cvFindNearestPoint2D

Finds the closest subdivision vertex to the given point.

```
CvSubdiv2DPoint* cvFindNearestPoint2D(
            CvSubdiv2D* subdiv,
            CvPoint2D32f pt );
```

subdiv Delaunay or another subdivision

pt Input point

The function is another function that locates the input point within the subdivision. It finds the subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using cvSubdiv2DLocate) is used as a starting point. The function returns a pointer to the found subdivision vertex.

cvSubdiv2DEdgeDst

Returns the edge destination.

```
CvSubdiv2DPoint* cvSubdiv2DEdgeDst(
            CvSubdiv2DEdge edge );
```

edge Subdivision edge (not a quad-edge)

The function returns the edge destination. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using the function cvCalcSubdivVoronoi2D.

cvSubdiv2DGetEdge

Returns one of the edges related to the given edge.

```
CvSubdiv2DEdge cvSubdiv2DGetEdge( CvSubdiv2DEdge edge, CvNextEdgeType
type );
```

```
#define cvSubdiv2DNextEdge( edge ) cvSubdiv2DGetEdge( edge, CV_NEXT_AROUND_ORG )
```

edge Subdivision edge (not a quad-edge)

type Specifies which of the related edges to return, one of the following:

- **CV_NEXT_AROUND_ORG** next around the edge origin (`eOnext` on the picture above if `e` is the input edge)
- **CV_NEXT_AROUND_DST** next around the edge vertex (`eDnext`)
- **CV_PREV_AROUND_ORG** previous around the edge origin (reversed `eRnext`)
- **CV_PREV_AROUND_DST** previous around the edge destination (reversed `eLnext`)
- **CV_NEXT_AROUND_LEFT** next around the left facet (`eLnext`)
- **CV_NEXT_AROUND_RIGHT** next around the right facet (`eRnext`)
- **CV_PREV_AROUND_LEFT** previous around the left facet (reversed `eOnext`)
- **CV_PREV_AROUND_RIGHT** previous around the right facet (reversed `eDnext`)

The function returns one of the edges related to the input edge.

---

cvSubdiv2DLocate

Returns the location of a point within a Delaunay triangulation.

```
CvSubdiv2DPointLocation cvSubdiv2DLocate(
           CvSubdiv2D* subdiv,
           CvPoint2D32f pt,
           CvSubdiv2DEdge* edge,
           CvSubdiv2DPoint** vertex=NULL );
```

subdiv Delaunay or another subdivision

pt The point to locate

edge The output edge the point falls onto or right to

vertex Optional output vertex double pointer the input point coinsides with

The function locates the input point within the subdivision. There are 5 cases:

- The point falls into some facet. The function returns `CV_PTLOC_INSIDE` and `*edge` will contain one of edges of the facet.
- The point falls onto the edge. The function returns `CV_PTLOC_ON_EDGE` and `*edge` will contain this edge.
- The point coincides with one of the subdivision vertices. The function returns `CV_PTLOC_VERTEX` and `*vertex` will contain a pointer to the vertex.
- The point is outside the subdivsion reference rectangle. The function returns `CV_PTLOC_OUTSIDE_RECT` and no pointers are filled.
- One of input arguments is invalid. A runtime error is raised or, if silent or "parent" error processing mode is selected, `CV_PTLOC_ERROR` is returnd.

---

cvSubdiv2DRotateEdge

Returns another edge of the same quad-edge.

```
CvSubdiv2DEdge cvSubdiv2DRotateEdge(
            CvSubdiv2DEdge edge,
            int rotate );
```

edge Subdivision edge (not a quad-edge)

rotate Specifies which of the edges of the same quad-edge as the input one to return, one of the following:

- **0** the input edge (`e` on the picture above if `e` is the input edge)
- **1** the rotated edge (`eRot`)
- **2** the reversed edge (reversed `e` (in green))
- **3** the reversed rotated edge (reversed `eRot` (in green))

The function returns one of the edges of the same quad-edge as the input edge.

---

cvSubdivDelaunay2DInsert

Inserts a single point into a Delaunay triangulation.

```
CvSubdiv2DPoint* cvSubdivDelaunay2DInsert(
            CvSubdiv2D* subdiv,
            CvPoint2D32f pt);
```

subdiv Delaunay subdivision created by the function cvCreateSubdivDelaunay2D

pt Inserted point

The function inserts a single point into a subdivision and modifies the subdivision topology appropriately. If a point with the same coordinates exists already, no new point is added. The function returns a pointer to the allocated point. No virtual point coordinates are calculated at this stage.

2.9 Object Detection

---

cvMatchTemplate

Compares a template against overlapped image regions.

```
void cvMatchTemplate(
            const CvArr* image,
            const CvArr* templ,
            CvArr* result,
            int method );
```

image Image where the search is running; should be 8-bit or 32-bit floating-point

templ Searched template; must be not greater than the source image and the same data type as the image

result A map of comparison results; single-channel 32-bit floating-point. If `image` is $W \times H$ and `templ` is $w \times h$ then `result` must be $(W - w + 1) \times (H - h + 1)$

method Specifies the way the template must be compared with the image regions (see below)

The function is similar to cvCalcBackProjectPatch. It slides through `image`, compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result`. Here are the formulas for the different comparison methods one may

use (I denotes `image`, T `template`, R `result`). The summation is done over template and/or the image patch: $x' = 0...w-1, y' = 0...h-1$

method=CV_TM_SQDIFF

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

method=CV_TM_SQDIFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCORR

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I(x+x',y+y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

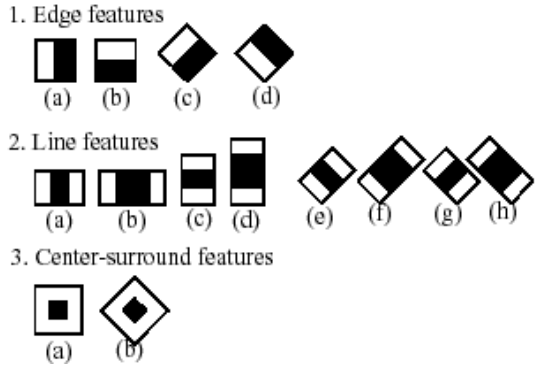
After the function finishes the comparison, the best matches can be found as global minimums (`CV_TM_SQDIFF`) or maximums (`CV_TM_CCORR` and `CV_TM_CCOEFF`) using the cvMinMaxLoc function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel).

Haar Feature-based Cascade Classifier for Object Detection

The object detector described below has been initially proposed by Paul Viola cvViola01 and improved by Rainer Lienhart cvLienhart02 . First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundred sample views of a particular object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.

After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a "1" if the region is likely to show the object (i.e., face/car), and "0" otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word "cascade" in the classifier name means that the resultant classifier consists of several simpler classifiers (*stages*) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word "boosted" means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different `boosting` techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifers, and are calculated as described below. The current algorithm uses the following Haar-like features:



The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in the case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the

differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and the cvIntegral description).

To see the object detector at work, have a look at the HaarFaceDetect demo.

The following reference is for the detection part only. There is a separate application called `haartraining` that can train a cascade of boosted classifiers from a set of samples. See `opencv/apps/haartraining` for details.

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade

Boosted Haar classifier structures.

```
#define CV_HAAR_FEATURE_MAX  3

/* a haar feature consists of 2-3 rectangles with appropriate weights */
typedef struct CvHaarFeature
{
    int  tilted;  /* 0 means up-right feature, 1 means 45--rotated feature */

    /* 2-3 rectangles with weights of opposite signs and
       with absolute values inversely proportional to the areas of the
       rectangles.  If rect[2].weight !=0, then
       the feature consists of 3 rectangles, otherwise it consists of 2 */
    struct
    {
        CvRect r;
        float weight;
    } rect[CV_HAAR_FEATURE_MAX];
}
CvHaarFeature;

/* a single tree classifier (stump in the simplest case) that returns the
   response for the feature at the particular image location (i.e. pixel
   sum over subrectangles of the window) and gives out a value depending
   on the response */
typedef struct CvHaarClassifier
{
    int count;  /* number of nodes in the decision tree */

    /* these are "parallel" arrays. Every index \texttt{i}
       corresponds to a node of the decision tree (root has 0-th index).

       left[i] - index of the left child (or negated index if the
         left child is a leaf)
```

```
       right[i] - index of the right child (or negated index if the
          right child is a leaf)
       threshold[i] - branch threshold. if feature responce is <= threshold,
                      left branch is chosen, otherwise right branch is chosen.
       alpha[i] - output value correponding to the leaf. */
    CvHaarFeature* haar_feature;
    float* threshold;
    int* left;
    int* right;
    float* alpha;
}
CvHaarClassifier;

/* a boosted battery of classifiers(=stage classifier):
   the stage classifier returns 1
   if the sum of the classifiers responses
   is greater than \texttt{threshold} and 0 otherwise */
typedef struct CvHaarStageClassifier
{
    int  count;  /* number of classifiers in the battery */
    float threshold; /* threshold for the boosted classifier */
    CvHaarClassifier* classifier; /* array of classifiers */

    /* these fields are used for organizing trees of stage classifiers,
       rather than just stright cascades */
    int next;
    int child;
    int parent;
}
CvHaarStageClassifier;

typedef struct CvHidHaarClassifierCascade CvHidHaarClassifierCascade;

/* cascade or tree of stage classifiers */
typedef struct CvHaarClassifierCascade
{
    int  flags; /* signature */
    int  count; /* number of stages */
    CvSize orig_window_size; /* original object size (the cascade is
                           trained for) */

    /* these two parameters are set by cvSetImagesForHaarClassifierCascade */
    CvSize real_window_size; /* current object size */
    double scale; /* current scale */
    CvHaarStageClassifier* stage_classifier; /* array of stage classifiers */
```

```
    CvHidHaarClassifierCascade* hid_cascade; /* hidden optimized
                        representation of the
                        cascade, created by
                   cvSetImagesForHaarClassifierCascade */
}
CvHaarClassifierCascade;
```

All the structures are used for representing a cascaded of boosted Haar classifiers. The cascade has the following hierarchical structure:

```
    Cascade:
        Stage,,1,,:
            Classifier,,11,,:
                Feature,,11,,
            Classifier,,12,,:
                Feature,,12,,
            ...
        Stage,,2,,:
            Classifier,,21,,:
                Feature,,21,,
            ...
        ...
```

The whole hierarchy can be constructed manually or loaded from a file or an embedded base using the function cvLoadHaarClassifierCascade.

---

cvLoadHaarClassifierCascade

Loads a trained cascade classifier from a file or the classifier database embedded in OpenCV.

```
CvHaarClassifierCascade* cvLoadHaarClassifierCascade(
           const char* directory,
           CvSize orig_window_size );
```

directory Name of the directory containing the description of a trained cascade classifier

orig_window_size Original size of the objects the cascade has been trained on. Note that it is not stored in the cascade and therefore must be specified separately

The function loads a trained cascade of haar classifiers from a file or the classifier database embedded in OpenCV. The base can be trained using the `haartraining` application (see opencv/apps/haartraining for details).

The function is obsolete. Nowadays object detection classifiers are stored in XML or YAML files, rather than in directories. To load a cascade from a file, use the cvLoad function.

---

cvHaarDetectObjects

Detects objects in the image.

```
typedef struct CvAvgComp
{
    CvRect rect; /* bounding rectangle for the object (average rectangle of a group) */
    int neighbors; /* number of neighbor rectangles in the group */
}
CvAvgComp;
```

```
CvSeq* cvHaarDetectObjects(
            const CvArr* image,
            CvHaarClassifierCascade* cascade,
            CvMemStorage* storage,
            double scale_factor=1.1,
            int min_neighbors=3,
            int flags=0,
            CvSize min_size=cvSize(0,
            0) );
```

image Image to detect objects in

cascade Haar classifier cascade in internal representation

storage Memory storage to store the resultant sequence of the object candidate rectangles

scale_factor The factor by which the search window is scaled between the subsequent scans, 1.1 means increasing window by 10%

min_neighbors Minimum number (minus 1) of neighbor rectangles that makes up an object. All the groups of a smaller number of rectangles than `min_neighbors`-1 are rejected. If `min_neighbors` is 0, the function does not any grouping at all and returns all the detected candidate rectangles, which may be useful if the user wants to apply a customized grouping procedure

flags Mode of operation. Currently the only flag that may be specified is CV_HAAR_DO_CANNY_PRUNING. If it is set, the function uses Canny edge detector to reject some image regions that contain too few or too much edges and thus can not contain the searched object. The particular threshold values are tuned for face detection and in this case the pruning speeds up the processing

min_size Minimum window size. By default, it is set to the size of samples the classifier has been trained on ($\sim 20 \times 20$ for face detection)

The function finds rectangular regions in the given image that are likely to contain objects the cascade has been trained for and returns those regions as a sequence of rectangles. The function scans the image several times at different scales (see cvSetImagesForHaarClassifierCascade). Each time it considers overlapping regions in the image and applies the classifiers to the regions using cvRunHaarClassifierCascade. It may also apply some heuristics to reduce number of analyzed regions, such as Canny prunning. After it has proceeded and collected the candidate rectangles (regions that passed the classifier cascade), it groups them and returns a sequence of average rectangles for each large enough group. The default parameters (scale_factor =1.1, min_neighbors =3, flags =0) are tuned for accurate yet slow object detection. For a faster operation on real video images the settings are: scale_factor =1.2, min_neighbors =2, flags =CV_HAAR_DO_CANNY_PRUNING, min_size =*minimum possible face size* (for example, $\sim$ 1/4 to 1/16 of the image area in the case of video conferencing).

```
#include "cv.h"
#include "highgui.h"

CvHaarClassifierCascade* load_object_detector( const char* cascade_path )
{
    return (CvHaarClassifierCascade*)cvLoad( cascade_path );
}

void detect_and_draw_objects( IplImage* image,
                              CvHaarClassifierCascade* cascade,
                              int do_pyramids )
{
    IplImage* small_image = image;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* faces;
    int i, scale = 1;

    /* if the flag is specified, down-scale the input image to get a
       performance boost w/o loosing quality (perhaps) */
    if( do_pyramids )
    {
        small_image = cvCreateImage( cvSize(image->width/2,image->height/2), IPL_DEPTH_8U,
```

```
        cvPyrDown( image, small_image, CV_GAUSSIAN_5x5 );
        scale = 2;
    }

    /* use the fastest variant */
    faces = cvHaarDetectObjects( small_image, cascade, storage, 1.2, 2, CV_HAAR_DO_CANNY_PR

    /* draw all the rectangles */
    for( i = 0; i < faces->total; i++ )
    {
        /* extract the rectanlges only */
        CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i );
        cvRectangle( image, cvPoint(face_rect.x*scale,face_rect.y*scale),
                     cvPoint((face_rect.x+face_rect.width)*scale,
                             (face_rect.y+face_rect.height)*scale),
                     CV_RGB(255,0,0), 3 );
    }

    if( small_image != image )
        cvReleaseImage( &small_image );
    cvReleaseMemStorage( &storage );
}

/* takes image filename and cascade path from the command line */
int main( int argc, char** argv )
{
    IplImage* image;
    if( argc==3 && (image = cvLoadImage( argv[1], 1 )) != 0 )
    {
        CvHaarClassifierCascade* cascade = load_object_detector(argv[2]);
        detect_and_draw_objects( image, cascade, 1 );
        cvNamedWindow( "test", 0 );
        cvShowImage( "test", image );
        cvWaitKey(0);
        cvReleaseHaarClassifierCascade( &cascade );
        cvReleaseImage( &image );
    }

    return 0;
}
```

---

cvSetImagesForHaarClassifierCascade

Assigns images to the hidden cascade.

```
void cvSetImagesForHaarClassifierCascade(
           CvHaarClassifierCascade* cascade,
           const CvArr* sum,
           const CvArr* sqsum,
           const CvArr* tilted_sum,
           double scale );
```

cascade Hidden Haar classifier cascade, created by cvCreateHidHaarClassifierCascade

sum Integral (sum) single-channel image of 32-bit integer format. This image as well as the two subsequent images are used for fast feature evaluation and brightness/contrast normalization. They all can be retrieved from input 8-bit or floating point single-channel image using the function cvIntegral

sqsum Square sum single-channel image of 64-bit floating-point format

tilted_sum Tilted sum single-channel image of 32-bit integer format

scale Window scale for the cascade. If `scale` =1, the original window size is used (objects of that size are searched) - the same size as specified in cvLoadHaarClassifierCascade (24x24 in the case of `default_face_cascade`), if `scale` =2, a two times larger window is used (48x48 in the case of default face cascade). While this will speed-up search about four times, faces smaller than 48x48 cannot be detected

The function assigns images and/or window scale to the hidden classifier cascade. If image pointers are NULL, the previously set images are used further (i.e. NULLs mean "do not change images"). Scale parameter has no such a "protection" value, but the previous value can be retrieved by the cvGetHaarClassifierCascadeScale function and reused again. The function is used to prepare cascade for detecting object of the particular size in the particular image. The function is called internally by cvHaarDetectObjects, but it can be called by the user if they are using the lower-level function cvRunHaarClassifierCascade.

---

cvReleaseHaarClassifierCascade

Releases the haar classifier cascade.

```
void cvReleaseHaarClassifierCascade(
           CvHaarClassifierCascade** cascade );
```

cascade Double pointer to the released cascade. The pointer is cleared by the function

The function deallocates the cascade that has been created manually or loaded using cvLoadHaarClassifierCascade or cvLoad.

---

cvRunHaarClassifierCascade

Runs a cascade of boosted classifiers at the given image location.

```
int cvRunHaarClassifierCascade(
            CvHaarClassifierCascade* cascade,
            CvPoint pt,
            int start_stage=0 );
```

cascade Haar classifier cascade

pt Top-left corner of the analyzed region. Size of the region is a original window size scaled by the currenly set scale. The current window size may be retrieved using the cvGetHaarClassifierCascadeWindowSize function

start_stage Initial zero-based index of the cascade stage to start from. The function assumes that all the previous stages are passed. This feature is used internally by cvHaarDetectObjects for better processor cache utilization

The function runs the Haar classifier cascade at a single image location. Before using this function the integral images and the appropriate scale (window size) should be set using cvSetImagesForHaarClassifierCascade. The function returns a positive value if the analyzed rectangle passed all the classifier stages (it is a candidate) and a zero or negative value otherwise.

2.10 Camera Calibration and 3D Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\, m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re-used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distorion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$
where $r^2 = x'^2 + y'^2$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

k_1, k_2, k_3 are radial distortion coefficients, p_1, p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed

or returned as

$$(k_1, k_2, p_1, p_2[, k_3])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters. *And they remain the same regardless of the captured image resolution.* That is, if, for example, a camera has been calibrated on images of 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x, f_y, c_x and c_y need to be scaled appropriately).

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspodences).
- Estimate the relative position and orientation of the stereo camera "heads" and compute the *rectification* transformation that makes the camera optical axes parallel.

---

cvCalcImageHomography

Calculates the homography matrix for an oblong planar object (e.g. arm).

```
void cvCalcImageHomography(
           float* line,
           CvPoint3D32f* center,
           float* intrinsic,
           float* homography );
```

line the main object axis direction (vector (dx,dy,dz))

center object center ((cx,cy,cz))

intrinsic intrinsic camera parameters (3x3 matrix)

homography output homography matrix (3x3)

The function calculates the homography matrix for the initial image transformation from image plane to the plane, defined by a 3D oblong object line (See __Figure 6-10__ in the OpenCV Guide 3D Reconstruction Chapter).

CalibrateCamera2

calibrateCamera Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
double cvCalibrateCamera2(
           const CvMat* objectPoints,
           const CvMat* imagePoints,
           const CvMat* pointCounts,
           CvSize imageSize,
           CvMat* cameraMatrix,
           CvMat* distCoeffs,
           CvMat* rvecs=NULL,
           CvMat* tvecs=NULL,
           int flags=0 );
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints The joint matrix of object points projections in the camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints` (=N).

imageSize Size of the image, used only to initialize the intrinsic camera matrix

cameraMatrix The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized before calling the function

distCoeffs The output 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

rvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of rotation vectors (see cvRodrigues2), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0..*M*-1)

tvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of translation vectors, estimated for each pattern view.

flags Different flags, may be 0 or combination of the following values:

CV_CALIB_USE_INTRINSIC_GUESS `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`imageSize` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate the extrinsic parameters. Use cvFindExtrinsicCameraParams2 instead.

CV_CALIB_FIX_PRINCIPAL_POINT The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.

CV_CALIB_FIX_ASPECT_RATIO The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix`.
When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see cvFindChessboardCorners). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).

2. The the initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using cvFindExtrinsicCameraParams2

3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the re-projection error, i.e. the total sum of squared distances between the observed feature points `imagePoints` and the projected (using the current estimates for camera parameters and the poses) object points `objectPoints`; see cvProjectPoints2.

The function returns the final re-projection error.

Note: if you're using a non-square (=non-NxN) grid and cv for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2-0.5, h/2-0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probaby used `patternSize=cvSize(rows,cols)`, but should use `patternSize=cvSize(cols,rows)` in cvFindChessboardCorners.

See also: cvFindChessboardCorners, cvFindExtrinsicCameraParams2, cv, cvStereoCalibrate, cvUndistort2

ComputeCorrespondEpilines

computeCorrespondEpilines For points in one image of a stereo pair, computes the corresponding epilines in the other image.

```
void cvComputeCorrespondEpilines(
           const CvMat* points,
           int whichImage,
           const CvMat* F,
           CvMat* lines);
```

points The input points. `2xN, Nx2, 3xN` or `Nx3` array (where `N` number of points). Multi-channel `1xN` or `Nx1` array is also acceptable

whichImage Index of the image (1 or 2) that contains the `points`

F The fundamental matrix that can be estimated using cvFindFundamentalMat or cvStereoRectify.

lines The output epilines, a `3xN` or `Nx3` array. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see cvFindFundamentalMat), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

ConvertPointsHomogeneous

convertPointsHomogeneous Convert points to/from homogeneous coordinates.

```
void cvConvertPointsHomogeneous(
           const CvMat* src,
           CvMat* dst );
```

src The input point array, `2xN, Nx2, 3xN, Nx3, 4xN or Nx4 (where N is the number of points)`. Multi-channel `1xN` or `Nx1` array is also acceptable

dst The output point array, must contain the same number of points as the input; The dimensionality must be the same, 1 less or 1 more than the input, and also within 2 to 4

The function converts 2D or 3D points from/to homogeneous coordinates, or simply copies or transposes the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$$x' = x/w$$
$$y' = y/w$$
$$z' = z/w \quad \text{(if output is 3D)}$$

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

Note because the function accepts a large variety of array layouts, it may report an error when input/output array dimensionality is ambiguous. It is always safe to use the function with number of points `N` ≥ 5, or to use multi-channel `Nx1` or `1xN` arrays.

---

cvCreatePOSITObject

Initializes a structure containing object information.

```
CvPOSITObject* cvCreatePOSITObject(
           CvPoint3D32f* points,
           int point_count );
```

points Pointer to the points of the 3D object model

point_count Number of object points

The function allocates memory for the object structure and computes the object inverse matrix.

The preprocessed object data is stored in the structure cvCvPOSITObject, internal for OpenCV, which means that the user cannot directly access the structure data. The user may only create this structure and pass its pointer to the function.

An object is defined as a set of points given in a coordinate system. The function cvPOSIT computes a vector that begins at a camera-related coordinate system center and ends at the `points[0]` of the object.

Once the work with a given object is finished, the function cvReleasePOSITObject must be called to free memory.

---

cvCreateStereoBMState

Creates block matching stereo correspondence structure.

```
CvStereoBMState* cvCreateStereoBMState( int preset=CV_STEREO_BM_BASIC,
int numberOfDisparities=0 );
```

preset ID of one of the pre-defined parameter sets. Any of the parameters can be overridden after creating the structure.

numberOfDisparities The number of disparities. If the parameter is 0, it is taken from the preset, otherwise the supplied value overrides the one from preset.

```
#define CV_STEREO_BM_BASIC 0
#define CV_STEREO_BM_FISH_EYE 1
#define CV_STEREO_BM_NARROW 2
```

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cvFindStereoCorrespondenceBM.

---

cvCreateStereoGCState

Creates the state of graph cut-based stereo correspondence algorithm.

```
CvStereoGCState* cvCreateStereoGCState( int numberOfDisparities, int
maxIters );
```

numberOfDisparities The number of disparities. The disparity search range will be `state->minDisparity`
$disparity <$ `state->minDisparity` $+$ `state->numberOfDisparities`

maxIters Maximum number of iterations. On each iteration all possible (or reasonable) alpha-expansions are tried. The algorithm may terminate earlier if it could not find an alpha-expansion that decreases the overall cost function value. See [**?**] for details.

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cvFindStereoCorrespondenceGC.

CvStereoBMState

The structure for block matching stereo correspondence algorithm.

```
typedef struct CvStereoBMState
{
    //pre filters (normalize input images):
    int       preFilterType; // 0 for now
    int       preFilterSize; // ~5x5..21x21
    int       preFilterCap;  // up to ~31
    //correspondence using Sum of Absolute Difference (SAD):
    int       SADWindowSize; // Could be 5x5..21x21
    int       minDisparity;  // minimum disparity (=0)
    int       numberOfDisparities; // maximum disparity - minimum disparity
```

```
    //post filters (knock out bad matches):
    int       textureThreshold; // areas with no texture are ignored
    float     uniquenessRatio;// filter out pixels if there are other close matches
                              // with different disparity
    int       speckleWindowSize; // the maximum area of speckles to remove
                                 // (set to 0 to disable speckle filtering)
    int       speckleRange; // acceptable range of disparity variation in each connected co
    // internal data
    ...
}
CvStereoBMState;
```

The block matching stereo correspondence algorithm, by Kurt Konolige, is very fast one-pass stereo matching algorithm that uses sliding sums of absolute differences between pixels in the left image and the pixels in the right image, shifted by some varying amount of pixels (from `minDisparity` to `minDisparity+numberOfDisparities`). On a pair of images WxH the algorithm computes disparity in `O(W*H*numberOfDisparities)` time. In order to improve quality and reability of the disparity map, the algorithm includes pre-filtering and post-filtering procedures.

Note that the algorithm searches for the corresponding blocks in x direction only. It means that the supplied stereo pair should be rectified. Vertical stereo layout is not directly supported, but in such a case the images could be transposed by user.

CvStereoGCState

The structure for graph cuts-based stereo correspondence algorithm

```
typedef struct CvStereoGCState
{
    int Ithreshold; // threshold for piece-wise linear data cost function (5 by default)
    int interactionRadius; // radius for smoothness cost function (1 by default; means Pott
    float K, lambda, lambda1, lambda2; // parameters for the cost function
                                       // (usually computed adaptively from the input data)
    int occlusionCost; // 10000 by default
    int minDisparity; // 0 by default; see CvStereoBMState
    int numberOfDisparities; // defined by user; see CvStereoBMState
    int maxIters; // number of iterations; defined by user.

    // internal buffers
    CvMat* left;
    CvMat* right;
    CvMat* dispLeft;
    CvMat* dispRight;
    CvMat* ptrLeft;
    CvMat* ptrRight;
```

```
    CvMat* vtxBuf;
    CvMat* edgeBuf;
}
CvStereoGCState;
```

The graph cuts stereo correspondence algorithm, described in [?] (as **KZ1**), is non-realtime stereo correpsondence algorithm that usually gives very accurate depth map with well-defined object boundaries. The algorithm represents stereo problem as a sequence of binary optimization problems, each of those is solved using maximum graph flow algorithm. The state structure above should not be allocated and initialized manually; instead, use cvCreateStereoGCState and then override necessary parameters if needed.

DecomposeProjectionMatrix

decomposeProjectionMatrix Decomposes the projection matrix into a rotation matrix and a camera matrix.

```
void cvDecomposeProjectionMatrix(
           const CvMat *projMatrix,
           CvMat *cameraMatrix,
           CvMat *rotMatrix,
           CvMat *transVect,
           CvMat *rotMatrX=NULL,
           CvMat *rotMatrY=NULL,
           CvMat *rotMatrZ=NULL,
           CvPoint3D64f *eulerAngles=NULL);
```

projMatrix The 3x4 input projection matrix P

cameraMatrix The output 3x3 camera matrix K

rotMatrix The output 3x3 external rotation matrix R

transVect The output 4x1 translation vector T

rotMatrX Optional 3x3 rotation matrix around x-axis

rotMatrY Optional 3x3 rotation matrix around y-axis

rotMatrZ Optional 3x3 rotation matrix around z-axis

eulerAngles Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on cvRQDecomp3x3.

DrawChessboardCorners

drawChessboardCorners Renders the detected chessboard corners.

```
void cvDrawChessboardCorners(
           CvArr* image,
           CvSize patternSize,
           CvPoint2D32f* corners,
           int count,
           int patternWasFound );
```

image The destination image; it must be an 8-bit color image

patternSize The number of inner corners per chessboard row and column. (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The array of corners detected

count The number of corners

patternWasFound Indicates whether the complete board was found $(\neq 0)$ or not $(= 0)$. One may just pass the return value cvFindChessboardCornersfindChessboardCorners here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

FindChessboardCorners

findChessboardCorners Finds the positions of the internal corners of the chessboard.

```
int cvFindChessboardCorners(
           const void* image,
           CvSize patternSize,
           CvPoint2D32f* corners,
           int* cornerCount=NULL,
           int flags=CV_CALIB_CB_ADAPTIVE_THRESH );
```

image Source chessboard view; it must be an 8-bit grayscale or color image

patternSize The number of inner corners per chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The output array of corners detected

cornerCount The output corner counter. If it is not NULL, it stores the number of corners found

flags Various operation flags, can be 0 or a combination of the following values:

- **CV_CALIB_CB_ADAPTIVE_THRESH** use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
- **CV_CALIB_CB_NORMALIZE_IMAGE** normalize the image gamma with cvEqualizeHist before applying fixed or adaptive thresholding.
- **CV_CALIB_CB_FILTER_QUADS** use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cvFindCornerSubPix.

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

FindExtrinsicCameraParams2

solvePnP Finds the object pose from the 3D-2D point correspondences

```
void cvFindExtrinsicCameraParams2(
           const CvMat* objectPoints,
           const CvMat* imagePoints,
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           CvMat* rvec,
           CvMat* tvec );
```

objectPoints The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points.

imagePoints The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points.

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$. If it is NULL, all of the distortion coefficients are set to 0

rvec The output rotation vector (see cvRodrigues2) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system

tvec The output translation vector

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using cvProjectPoints2) `objectPoints`.

FindFundamentalMat

findFundamentalMat Calculates the fundamental matrix from the corresponding points in two images.

```
int cvFindFundamentalMat(
           const CvMat* points1,
           const CvMat* points2,
           CvMat* fundamentalMatrix,
           int method=CV_FM_RANSAC,
           double param1=1.,
           double param2=0.99,
           CvMat* status=NULL);
```

points1 Array of `N` points from the first image. It can be `2xN, Nx2, 3xN` or `Nx3` 1-channel array or `1xN` or `Nx1` 2- or 3-channel array . The point coordinates should be floating-point (single or double precision)

points2 Array of the second image points of the same size and format as `points1`

fundamentalMatrix The output fundamental matrix or matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices)

method Method for computing the fundamental matrix

CV_FM_7POINT for a 7-point algorithm. $N = 7$

CV_FM_8POINT for an 8-point algorithm. $N \geq 8$

CV_FM_RANSAC for the RANSAC algorithm. $N \geq 8$

CV_FM_LMEDS for the LMedS algorithm. $N \geq 8$

param1 The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise

param2 The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct

status The optional output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1's

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the number of fundamental matrices found (1 or 3) and 0, if no matrix is found . Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to cvComputeCorrespondEpilines that finds the epipolar lines corresponding to the specified points. It can also be passed to cvStereoRectifyUncalibrated to compute the rectification transformation.

```
int point_count = 100;
CvMat* points1;
CvMat* points2;
CvMat* status;
CvMat* fundamental_matrix;

points1 = cvCreateMat(1,point_count,CV_32FC2);
points2 = cvCreateMat(1,point_count,CV_32FC2);
status = cvCreateMat(1,point_count,CV_8UC1);

/* Fill the points here ... */
for( i = 0; i < point_count; i++ )
{
    points1->data.fl[i*2] = <x,,1,i,,>;
    points1->data.fl[i*2+1] = <y,,1,i,,>;
    points2->data.fl[i*2] = <x,,2,i,,>;
    points2->data.fl[i*2+1] = <y,,2,i,,>;
}

fundamental_matrix = cvCreateMat(3,3,CV_32FC1);
int fm_count = cvFindFundamentalMat( points1,points2,fundamental_matrix,
                                     CV_FM_RANSAC,1.0,0.99,status );
```

FindHomography

findHomography Finds the perspective transformation between two planes.

```
void cvFindHomography(
           const CvMat* srcPoints,
           const CvMat* dstPoints,
           CvMat* H
```

```
            int method=0,
            double ransacReprojThreshold=0,
            CvMat* status=NULL);
```

srcPoints Coordinates of the points in the original plane, 2xN, Nx2, 3xN or Nx3 1-channel array (the latter two are for representation in homogeneous coordinates), where N is the number of points. 1xN or Nx1 2- or 3-channel array can also be passed.

dstPoints Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 1-channel, or 1xN or Nx1 2- or 3-channel array.

H The output 3x3 homography matrix

method The method used to computed homography matrix; one of the following:

0 a regular method using all the points

CV_RANSAC RANSAC-based robust method

CV_LMEDS Least-Median robust method

ransacReprojThreshold The maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.

status The optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). *Note that the input mask values are ignored.*

The function finds the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}} \right)^2$$

is minimized. If the parameter `method` is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, `RANSAC` and `LMeDS`, try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection error for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method `RANSAC` can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method `LMeDS` does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features, where can be only some small noise present, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized so that $h_{33} = 1$.

See also: cvGetAffineTransform, cvGetPerspectiveTransform, cvEstimateRigidMotion, cvWarpPerspective, cvPerspectiveTransform

---

cvFindStereoCorrespondenceBM

Computes the disparity map using block matching algorithm.

```
void cvFindStereoCorrespondenceBM(
           const CvArr* left,
           const CvArr* right,
           CvArr* disparity,
           CvStereoBMState* state );
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

disparity The output single-channel 16-bit signed, or 32-bit floating-point disparity map of the same size as input images. In the first case the computed disparities are represented as

fixed-point numbers with 4 fractional bits (i.e. the computed disparity values are multiplied by 16 and rounded to integers).

state Stereo correspondence structure.

The function cvFindStereoCorrespondenceBM computes disparity map for the input rectified stereo pair. Invalid pixels (for which disparity can not be computed) are set to `state->minDisparity - 1` (or to `(state->minDisparity-1)*16` in the case of 16-bit fixed-point disparity map)

---

cvFindStereoCorrespondenceGC

Computes the disparity map using graph cut-based algorithm.

```
void cvFindStereoCorrespondenceGC(
           const CvArr* left,
           const CvArr* right,
           CvArr* dispLeft,
           CvArr* dispRight,
           CvStereoGCState* state,
           int useDisparityGuess = CV_DEFAULT(0) );
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

dispLeft The optional output single-channel 16-bit signed left disparity map of the same size as input images.

dispRight The optional output single-channel 16-bit signed right disparity map of the same size as input images.

state Stereo correspondence structure.

useDisparityGuess If the parameter is not zero, the algorithm will start with pre-defined disparity maps. Both dispLeft and dispRight should be valid disparity maps. Otherwise, the function starts with blank disparity maps (all pixels are marked as occlusions).

The function computes disparity maps for the input rectified stereo pair. Note that the left disparity image will contain values in the following range:

$$\texttt{-state->numberOfDisparities-state->minDisparity} < dispLeft(x,y) \leq \texttt{-state->minDisparity}$$

or

$$dispLeft(x,y) == \texttt{CV\_STEREO\_GC\_OCCLUSION}$$

and for the right disparity image the following will be true:

$$\texttt{state->minDisparity} \leq dispRight(x,y) < \texttt{state->minDisparity+state->numberOfDisparities}$$

or

$$dispRight(x,y) == \texttt{CV\_STEREO\_GC\_OCCLUSION}$$

that is, the range for the left disparity image will be inversed, and the pixels for which no good match has been found, will be marked as occlusions.

Here is how the function can be called:

```
// image_left and image_right are the input 8-bit single-channel images
// from the left and the right cameras, respectively
CvSize size = cvGetSize(image_left);
CvMat* disparity_left = cvCreateMat( size.height, size.width, CV_16S );
CvMat* disparity_right = cvCreateMat( size.height, size.width, CV_16S );
CvStereoGCState* state = cvCreateStereoGCState( 16, 2 );
cvFindStereoCorrespondenceGC( image_left, image_right,
    disparity_left, disparity_right, state, 0 );
cvReleaseStereoGCState( &state );
// now process the computed disparity images as you want ...
```

and this is the output left disparity image computed from the well-known Tsukuba stereo pair and multiplied by -16 (because the values in the left disparity images are usually negative):

```
CvMat* disparity_left_visual = cvCreateMat( size.height, size.width, CV_8U );
cvConvertScale( disparity_left, disparity_left_visual, -16 );
cvSave( "disparity.pgm", disparity_left_visual );
```

GetOptimalNewCameraMatrix

getOptimalNewCameraMatrix Returns the new camera matrix based on the free scaling parameter

```
void cvGetOptimalNewCameraMatrix(
           const CvMat* cameraMatrix, const CvMat* distCoeffs,
           CvSize imageSize, double alpha,
           CvMat* newCameraMatrix,
           CvSize newImageSize=cvSize(0,0),
           CvRect* validPixROI=0 );
```

cameraMatrix The input camera matrix

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

imageSize The original image size

alpha The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see cvStereoRectify

newCameraMatrix The output new camera matrix.

newImageSize The image size after rectification. By default it will be set to `imageSize`.

validPixROI The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in cvStereoRectify

The function computes the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0`, keep all the original image pixels if there is valuable information in the corners `alpha=1`, or get something in between. When `alpha>0`, the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to cvInitUndistortRectifyMap to produce the maps for cvRemap.

InitIntrinsicParams2D

initCameraMatrix2D Finds the initial camera matrix from the 3D-2D point correspondences

```
void cvInitIntrinsicParams2D(
           const CvMat* objectPoints,
           const CvMat* imagePoints,
           const CvMat* npoints, CvSize imageSize,
           CvMat* cameraMatrix,
           double aspectRatio=1.);
```

objectPoints The joint array of object points; see cvCalibrateCamera2

imagePoints The joint array of object point projections; see cvCalibrateCamera2

npoints The array of point counts; see cvCalibrateCamera2

imageSize The image size in pixels; used to initialize the principal point

cameraMatrix The output camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

aspectRatio If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

InitUndistortMap

Computes an undistortion map.

```
void cvInitUndistortMap(
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           CvArr* map1,
           CvArr* map2 );
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function is a simplified variant of cvInitUndistortRectifyMap where the rectification transformation `R` is identity matrix and `newCameraMatrix=cameraMatrix`.

InitUndistortRectifyMap

initUndistortRectifyMap Computes the undistortion and rectification transformation map.

```
void cvInitUndistortRectifyMap(
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
```

```
            const CvMat* R,
            const CvMat* newCameraMatrix,
            CvArr* map1,
            CvArr* map2 );
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

R The optional rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cvStereoRectify can be passed here. If the matrix is NULL , the identity transformation is assumed

newCameraMatrix The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for cvRemap. The undistorted image will look like the original, as if it was captured with a camera with camera matrix `=newCameraMatrix` and zero distortion. In the case of monocular camera `newCameraMatrix` is usually equal to `cameraMatrix`, or it can be computed by cvGetOptimalNewCameraMatrix for a better control over scaling. In the case of stereo camera `newCameraMatrix` is normally set to `P1` or `P2` computed by cvStereoRectify.

Also, this new camera will be oriented differently in the coordinate space, according to `R`. That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images become horizontal and have the same y- coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by cvRemap. That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$
\begin{array}{l}
x \leftarrow (u - {c'}_x)/{f'}_x \\
y \leftarrow (v - {c'}_y)/{f'}_y \\
[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\
x' \leftarrow X/W \\
y' \leftarrow Y/W \\
x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
map_x(u,v) \leftarrow x" f_x + c_x \\
map_y(u,v) \leftarrow y" f_y + c_y
\end{array}
$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after cvStereoRectify, which in its turn is called after cvStereoCalibrate. But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using cvStereoRectifyUncalibrated. For each camera the function computes homography `H` as the rectification transformation in pixel domain, not a rotation matrix `R` in 3D space. The `R` can be computed from `H` as

$$\texttt{R} = \texttt{cameraMatrix}^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

---

cvPOSIT

Implements the POSIT algorithm.

```
void cvPOSIT(
           CvPOSITObject* posit_object,
           CvPoint2D32f* imagePoints,
           double focal_length,
           CvTermCriteria criteria,
           CvMatr32f rotationMatrix,
           CvVect32f translation_vector );
```

posit_object Pointer to the object structure

imagePoints Pointer to the object points projections on the 2D image plane

focal_length Focal length of the camera used

criteria Termination criteria of the iterative POSIT algorithm

rotationMatrix Matrix of rotations

translation_vector Translation vector

The function implements the POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using the camera calibration functions. At every iteration of the algorithm a new perspective projection of the estimated pose is computed.

Difference norm between two projections is the maximal distance between corresponding points. The parameter `criteria.epsilon` serves to stop the algorithm if the difference is small.

ProjectPoints2

projectPoints Project 3D points on to an image plane.

```
void cvProjectPoints2(
           const CvMat* objectPoints,
           const CvMat* rvec,
           const CvMat* tvec,
           const CvMat* cameraMatrix,
           const CvMat* distCoeffs,
           CvMat* imagePoints,
           CvMat* dpdrot=NULL,
           CvMat* dpdt=NULL,
           CvMat* dpdf=NULL,
           CvMat* dpdc=NULL,
           CvMat* dpddist=NULL );
```

objectPoints The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel , where N is the number of points in the view

rvec The rotation vector, see cvRodrigues2

tvec The translation vector

cameraMatrix The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$. If it is NULL , all of the distortion coefficients are considered 0's

imagePoints The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel

dpdrot Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector

dpdt Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector

dpdf Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y

dpdc Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y

dpddist Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in cvCalibrateCamera2, cvFindExtrinsicCameraParams2 and cvStereoCalibrate. The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

ReprojectImageTo3D

reprojectImageTo3D Reprojects disparity image to 3D space.

```
void cvReprojectImageTo3D( const CvArr* disparity,
           CvArr* _3dImage, const CvMat* Q,
           int handleMissingValues=0);
```

disparity The input single-channel 16-bit signed or 32-bit floating-point disparity image

_3dImage The output 3-channel floating-point image of the same size as `disparity`. Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)`, computed from the disparity map.

Q The 4×4 perspective transformation matrix that can be obtained with cvStereoRectify

handleMissingValues If true, when the pixels with the minimal disparity (that corresponds to the ouliers; see cvFindStereoCorrespondenceBM) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \mathtt{Q} * [x\ y\ \mathtt{disparity}(x,y)\ 1]^T$$
$$\mathtt{\_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by cvStereoRectify. To reproject a sparse set of points (x,y,d),... to 3D space, use cvPerspectiveTransform.

RQDecomp3x3

RQDecomp3x3 Computes the 'RQ' decomposition of 3x3 matrices.

```
void cvRQDecomp3x3(
            const CvMat *M,
            CvMat *R,
            CvMat *Q,
            CvMat *Qx=NULL,
            CvMat *Qy=NULL,
            CvMat *Qz=NULL,
            CvPoint3D64f *eulerAngles=NULL);
```

M The 3x3 input matrix

R The output 3x3 upper-triangular matrix

Q The output 3x3 orthogonal matrix

Qx Optional 3x3 rotation matrix around x-axis

Qy Optional 3x3 rotation matrix around y-axis

Qz Optional 3x3 rotation matrix around z-axis

eulerAngles Optional three Euler angles of rotation

The function computes a RQ decomposition using the given rotations. This function is used in cvDecomposeProjectionMatrix to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

---

cvReleasePOSITObject

Deallocates a 3D object structure.

```
void cvReleasePOSITObject(
           CvPOSITObject** posit_object );
```

posit_object Double pointer to `CvPOSIT` structure

The function releases memory previously allocated by the function cvCreatePOSITObject.

---

cvReleaseStereoBMState

Releases block matching stereo correspondence structure.

```
void cvReleaseStereoBMState( CvStereoBMState** state );
```

state Double pointer to the released structure.

The function releases the stereo correspondence structure and all the associated internal buffers.

---

cvReleaseStereoGCState

Releases the state structure of the graph cut-based stereo correspondence algorithm.

```
void cvReleaseStereoGCState( CvStereoGCState** state );
```

state Double pointer to the released structure.

The function releases the stereo correspondence structure and all the associated internal buffers.

Rodrigues2

Rodrigues Converts a rotation matrix to a rotation vector or vice versa.

```
int cvRodrigues2(
            const CvMat* src,
            CvMat* dst,
            CvMat* jacobian=0 );
```

src The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)

dst The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively

jacobian Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like cvCalibrateCamera2, cvStereoCalibrate or cvFindExtrinsicCameraParams2.

StereoCalibrate

stereoCalibrate Calibrates stereo camera.

```
double cvStereoCalibrate(
           const CvMat* objectPoints,
           const CvMat* imagePoints1,
           const CvMat* imagePoints2,
           const CvMat* pointCounts,
           CvMat* cameraMatrix1,
           CvMat* distCoeffs1,
           CvMat* cameraMatrix2,
           CvMat* distCoeffs2,
           CvSize imageSize,
           CvMat* R,
           CvMat* T,
           CvMat* E=0,
           CvMat* F=0,
           CvTermCriteria term_crit=cvTermCriteria(
           CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,30,1e-6),
           int flags=CV_CALIB_FIX_INTRINSIC );
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints1 The joint matrix of object points projections in the first camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

imagePoints2 The joint matrix of object points projections in the second camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints*` (=N).

cameraMatrix1 The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of CV_CALIB_USE_INTRINSIC_GUESS, CV_CALIB_FIX_ASPECT_RATIO, CV_CALIB_FIX_INTRINSIC or CV_CALIB_FIX_FOCAL_LENGTH are specified, some or all of the matrices' components must be initialized; see the flags description

distCoeffs1 The input/output lens distortion coefficients for the first camera, 4x1, 5x1, 1x4 or 1x5 floating-point vectors $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$, $j = 0, 1$. If any of CV_CALIB_FIX_K1, CV_CALIB_FIX_K2 or CV_CALIB_FIX_K3 is specified, then the corresponding elements of the distortion coefficients must be initialized.

cameraMatrix2 The input/output second camera matrix, as cameraMatrix1.

distCoeffs2 The input/output lens distortion coefficients for the second camera, as distCoeffs1.

imageSize Size of the image, used only to initialize intrinsic camera matrix.

R The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The output translation vector between the cameras' coordinate systems.

E The optional output essential matrix.

F The optional output fundamental matrix.

term_crit The termination criteria for the iterative optimization algorithm.

flags Different flags, may be 0 or combination of the following values:

- **CV_CALIB_FIX_INTRINSIC** If it is set, cameraMatrix?, as well as distCoeffs? are fixed, so that only R, T, E and F are estimated.
- **CV_CALIB_USE_INTRINSIC_GUESS** The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.
- **CV_CALIB_FIX_PRINCIPAL_POINT** The principal points are fixed during the optimization.
- **CV_CALIB_FIX_FOCAL_LENGTH** $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.
- **CV_CALIB_FIX_ASPECT_RATIO** $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.
- **CV_CALIB_SAME_FOCAL_LENGTH** Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$
- **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients for each camera are set to zeros and fixed there.

CV_CALIB_FIX_K1, CV_CALIB_FIX_K2, CV_CALIB_FIX_K3 Fixes the corresponding radial distortion coefficient (the coefficient must be passed to the function)

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientatation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with cvFindExtrinsicCameraParams2), obviously, those poses will relate to each other, i.e. given (R_1, T_1) it should be possible to compute (R_2, T_2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector T: $T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform full calibration of each of the 2 cameras. However, because of the high dimensionality of the parameter space and noise in the input data the function can diverge from the correct solution. Thus, if intrinsic parameters can be estimated with high accuracy for each of the cameras individually (e.g. using cvCalibrateCamera2), it is recommended to do so and then pass `CV_CALIB_FIX_INTRINSIC` flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are estimated at once, it makes sense to restrict some parameters, e.g. pass `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which are usually reasonable assumptions.

Similarly to cvCalibrateCamera2, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

StereoRectify

stereoRectify Computes rectification transforms for each head of a calibrated stereo camera.

```
void cvStereoRectify(
           const CvMat* cameraMatrix1, const CvMat* cameraMatrix2,
           const CvMat* distCoeffs1, const CvMat* distCoeffs2,
           CvSize imageSize, const CvMat* R, const CvMat* T,
           CvMat* R1, CvMat* R2, CvMat* P1, CvMat* P2,
           CvMat* Q=0, int flags=CV_CALIB_ZERO_DISPARITY,
           double alpha=-1, CvSize newImageSize=cvSize(0,0),
           CvRect* roi1=0, CvRect* roi2=0);
```

cameraMatrix1, cameraMatrix2 The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs1, distCoeffs2 The input distortion coefficients for each camera, ${k_1}^{(j)}, {k_2}^{(j)}, {p_1}^{(j)}, {p_2}^{(j)}[, {k_3}^{(j)}]$

imageSize Size of the image used for stereo calibration.

R The rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The translation vector between the cameras' coordinate systems.

R1, R2 The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.

P1, P2 The output 3×4 projection matrices in the new (rectified) coordinate systems.

Q The output 4×4 disparity-to-depth mapping matrix, see cv.

flags The operation flags; may be 0 or `CV_CALIB_ZERO_DISPARITY`. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.

alpha The free scaling parameter. If it is -1 , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.

newImageSize The new image resolution after rectification. The same size should be passed to cvInitUndistortRectifyMap, see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.

roi1, roi2 The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0`, the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by cv and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

1. Horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. Vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

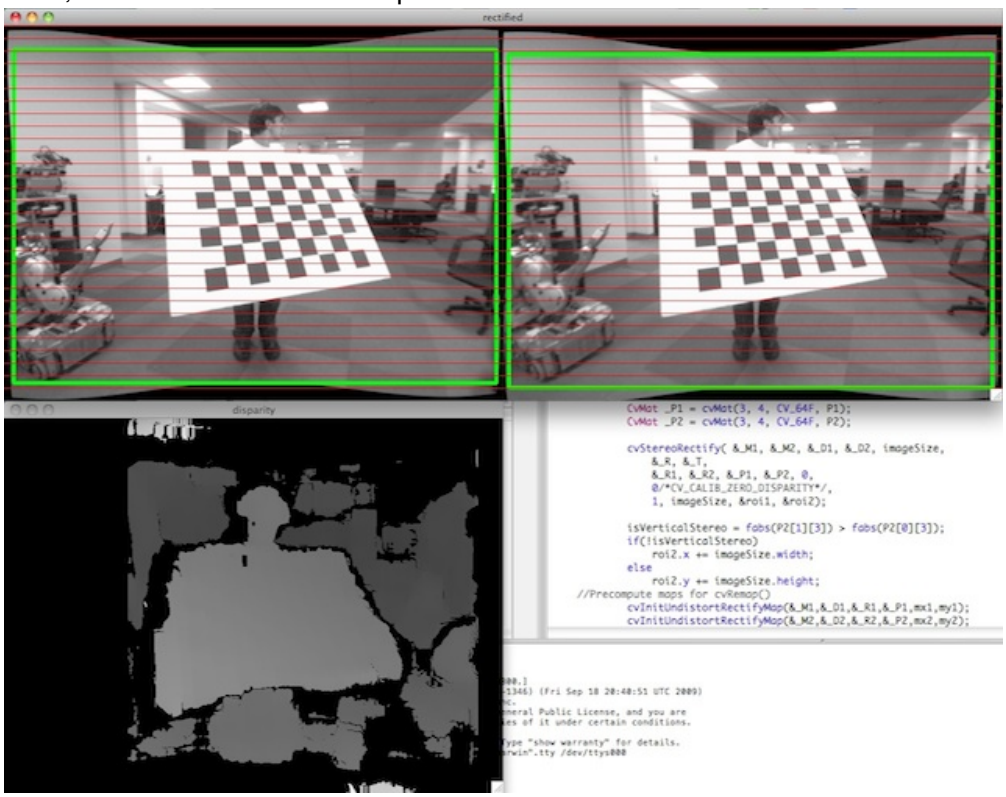
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of `P1` and `P2` will effectively be the new "rectified" camera matrices. The matrices, together with `R1` and `R2`, can then be passed to cvInitUndistortRectifyMap to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



StereoRectifyUncalibrated

stereoRectifyUncalibrated Computes rectification transform for uncalibrated stereo camera.

```
void cvStereoRectifyUncalibrated(
           const CvMat* points1,
           const CvMat* points2,
           const CvMat* F,
           CvSize imageSize,
           CvMat* H1,
           CvMat* H2,
           double threshold=5 );
```

points1, points2 The 2 arrays of corresponding 2D points. The same formats as in cvFindFundamentalMat are supported

F The input fundamental matrix. It can be computed from the same set of point pairs using cvFindFundamentalMat.

imageSize Size of the image.

H1, H2 The output rectification homography matrices for the first and for the second images.

threshold The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from cvStereoRectify is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the algorithm [**?**].

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using cvCalibrateCamera2 and then the images can be corrected using cvUndistort2, or just the point coordinates can be corrected with cvUndistortPoints.

Undistort2

undistort Transforms an image to compensate for lens distortion.

```
void cvUndistort2(
            const CvArr* src,
            CvArr* dst,
            const CvMat* cameraMatrix,
            const CvMat* distCoeffs,
            const CvMat* newCameraMatrix=0 );
```

src The input (distorted) image

dst The output (corrected) image; will have the same size and the same type as `src`

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The vector of distortion coefficients, $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of cvInitUndistortRectifyMap (with unity `R`) and cvRemap (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix`. You can use cvGetOptimalNewCameraMatrix to compute the appropriate `newCameraMatrix`, depending on your requirements.

The camera matrix and the distortion parameters can be determined using cvCalibrateCamera2. If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

UndistortPoints

undistortPoints Computes the ideal point coordinates from the observed point coordinates.

```
void cvUndistortPoints(
            const CvMat* src,
            CvMat* dst,
```

```
            const CvMat* cameraMatrix,
            const CvMat* distCoeffs,
            const CvMat* R=NULL,
            const CvMat* P=NULL);
```

src The observed point coordinates, same format as `imagePoints` in cvProjectPoints2

dst The output ideal point coordinates, after undistortion and reverse perspective transformation , same format as `src` .

cameraMatrix The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The vector of distortion coefficients, $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$

R The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv can be passed here. If the matrix is empty, the identity transformation is used

P The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2`, computed by cv can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to cvUndistort2 and cvInitUndistortRectifyMap, but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to cvProjectPoints2 (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used both for a stereo camera head or for monocular camera (when R is NULL).

Chapter 3

cvaux. Extra Computer Vision Functionality

Chapter 4

highgui. High-level GUI and Media I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and ”remember” their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

4.1 User Interface

---

cvConvertImage

Converts one image to another with an optional vertical flip.

```
void cvConvertImage( const CvArr* src, CvArr* dst, int flags=0 );
```

src Source image.

dst Destination image. Must be single-channel or 3-channel 8-bit image.

flags The operation flags:

CV_CVTIMG_FLIP Flips the image vertically

CV_CVTIMG_SWAP_RB Swaps the red and blue channels. In OpenCV color images have `BGR` channel order, however on some systems the order needs to be reversed before displaying the image (ShowImage does this automatically).

The function `cvConvertImage` converts one image to another and flips the result vertically if desired. The function is used by ShowImage .

---

cvCreateTrackbar

Creates a trackbar and attaches it to the specified window

```
int cvCreateTrackbar(
            const char* trackbarName,
            const char* windowName,
            int* value,
            int count,
            CvTrackbarCallback onChange );
```

trackbarName Name of the created trackbar.

windowName Name of the window which will be used as a parent for created trackbar.

value Pointer to an integer variable, whose value will reflect the position of the slider. Upon creation, the slider position is defined by this variable.

count Maximal position of the slider. Minimal position is always 0.

onChange Pointer to the function to be called every time the slider changes position. This function should be prototyped as `void Foo(int);` Can be NULL if callback is not required.

The function `cvCreateTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable to be syncronized with trackbar position and specifies a callback function to be called on trackbar position change. The created trackbar is displayed on the top of the given window.

```
CV_EXTERN_C_FUNCPTR( void (*CvTrackbarCallback)(int pos) );
```

cvDestroyAllWindows

Destroys all of the HighGUI windows.

```
void cvDestroyAllWindows(void);
```

The function `cvDestroyAllWindows` destroys all of the opened HighGUI windows.

cvDestroyWindow

Destroys a window.

```
void cvDestroyWindow( const char* name );
```

`name` Name of the window to be destroyed.

The function `cvDestroyWindow` destroys the window with the given name.

cvGetTrackbarPos

Returns the trackbar position.

```
int cvGetTrackbarPos(
           const char* trackbarName,
           const char* windowName );
```

`trackbarName` Name of the trackbar.

`windowName` Name of the window which is the parent of the trackbar.

The function `cvGetTrackbarPos` returns the current position of the specified trackbar.

cvGetWindowHandle

Gets the window's handle by its name.

```
void* cvGetWindowHandle( const char* name );
```

name Name of the window .

The function `cvGetWindowHandle` returns the native window handle (HWND in case of Win32 and GtkWidget in case of GTK+).

---

cvGetWindowName

Gets the window's name by its handle.

```
const char* cvGetWindowName( void* windowHandle );
```

windowHandle Handle of the window.

The function `cvGetWindowName` returns the name of the window given its native handle (HWND in case of Win32 and GtkWidget in case of GTK+).

---

cvInitSystem

Initializes HighGUI.

```
int cvInitSystem( int argc, char** argv );
```

argc Number of command line arguments

argv Array of command line arguments

The function `cvInitSystem` initializes HighGUI. If it wasn't called explicitly by the user before the first window was created, it is called implicitly then with `argc=0`, `argv=NULL`. Under Win32 there is no need to call it explicitly. Under X Window the arguments may be used to customize a look of HighGUI windows and controls.

cvMoveWindow

Sets the position of the window.

```
void cvMoveWindow( const char* name, int x, int y );
```

name Name of the window to be moved.

x New x coordinate of the top-left corner

y New y coordinate of the top-left corner

The function `cvMoveWindow` changes the position of the window.

cvNamedWindow

Creates a window.

```
int cvNamedWindow( const char* name, int flags );
```

name Name of the window in the window caption that may be used as a window identifier.

flags Flags of the window. Currently the only supported flag is `CV_WINDOW_AUTOSIZE`. If this is set, window size is automatically adjusted to fit the displayed image (see ShowImage), and the user can not change the window size manually.

The function `cvNamedWindow` creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing.

cvResizeWindow

Sets the window size.

```
void cvResizeWindow( const char* name, int width, int height );
```

name Name of the window to be resized.

width New width

height New height

The function `cvResizeWindow` changes the size of the window.

---

cvSetMouseCallback

Assigns callback for mouse events.

```
void cvSetMouseCallback( const char* windowName, CvMouseCallback
onMouse, void* param=NULL );
```

windowName Name of the window.

onMouse Pointer to the function to be called every time a mouse event occurs in the specified window. This function should be prototyped as `void Foo(int event, int x, int y, int flags, void* param);` where `event` is one of `CV_EVENT_*`, `x` and `y` are the coordinates of the mouse pointer in image coordinates (not window coordinates), `flags` is a combination of `CV_EVENT_FLAG_*`, and `param` is a user-defined parameter passed to the `cvSetMouseCallback` function call.

param User-defined parameter to be passed to the callback function.

The function `cvSetMouseCallback` sets the callback function for mouse events occuring within the specified window.

The `event` parameter is one of:

CV_EVENT_MOUSEMOVE Mouse movement

CV_EVENT_LBUTTONDOWN Left button down

CV_EVENT_RBUTTONDOWN Right button down

CV_EVENT_MBUTTONDOWN Middle button down

CV_EVENT_LBUTTONUP Left button up

CV_EVENT_RBUTTONUP Right button up

CV_EVENT_MBUTTONUP Middle button up

CV_EVENT_LBUTTONDBLCLK Left button double click

CV_EVENT_RBUTTONDBLCLK Right button double click

CV_EVENT_MBUTTONDBLCLK Middle button double click

The `flags` parameter is a combination of :

CV_EVENT_FLAG_LBUTTON Left button pressed

CV_EVENT_FLAG_RBUTTON Right button pressed

CV_EVENT_FLAG_MBUTTON Middle button pressed

CV_EVENT_FLAG_CTRLKEY Control key pressed

CV_EVENT_FLAG_SHIFTKEY Shift key pressed

CV_EVENT_FLAG_ALTKEY Alt key pressed

---

cvSetTrackbarPos

Sets the trackbar position.

```
void cvSetTrackbarPos(
           const char* trackbarName,
           const char* windowName,
           int pos );
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of trackbar.

pos New position.

The function `cvSetTrackbarPos` sets the position of the specified trackbar.

---

cvShowImage

Displays the image in the specified window

```
void cvShowImage( const char* name, const CvArr* image );
```

name Name of the window.

image Image to be shown.

The function `cvShowImage` displays the image in the specified window. If the window was created with the `CV_WINDOW_AUTOSIZE` flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

---

cvWaitKey

Waits for a pressed key.

```
int cvWaitKey( int delay=0 );
```

delay Delay in milliseconds.

The function `cvWaitKey` waits for key event infinitely ($\texttt{delay} <= 0$) or for `delay` milliseconds. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

4.2 Reading and Writing Images and Video

cvLoadImage

Loads an image from a file.

```
IplImage* cvLoadImage(
           const char* filename,
           int iscolor=CV_LOAD_IMAGE_COLOR );
```

```
#define CV_LOAD_IMAGE_COLOR       1
#define CV_LOAD_IMAGE_GRAYSCALE   0
#define CV_LOAD_IMAGE_UNCHANGED  -1
```

filename Name of file to be loaded.

iscolor Specific color type of the loaded image: if > 0, the loaded image is forced to be a 3-channel color image; if 0, the loaded image is forced to be grayscale; if < 0, the loaded image will be loaded as is (note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB).

The function `cvLoadImage` loads an image from the specified file and returns the pointer to the loaded image. Currently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

cvSaveImage

Saves an image to a specified file.

```
int cvSaveImage( const char* filename, const CvArr* image );
```

filename Name of the file.

image Image to be saved.

The function cvSaveImage saves the image to the specified file. The image format is chosen based on the filename extension, see LoadImage . Only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use cvCvtScale and cvCvtColor to convert it before saving, or use universal cvSave to save the image to XML or YAML format.

CvCapture

Video capturing structure.

```
typedef struct CvCapture CvCapture;
```

The structure CvCapture does not have a public interface and is used only as a parameter for video capturing functions.

---

cvCaptureFromCAM

Initializes capturing a video from a camera.

```
CvCapture* cvCaptureFromCAM( int index );
```

index Index of the camera to be used. If there is only one camera or it does not matter what camera is used -1 may be passed.

The function cvCaptureFromCAM allocates and initializes the CvCapture structure for reading a video stream from the camera. Currently two camera interfaces can be used on Windows: Video for Windows (VFW) and Matrox Imaging Library (MIL); and two on Linux: V4L and FireWire (IEEE1394).

To release the structure, use ReleaseCapture .

cvCaptureFromFile

Initializes capturing a video from a file.

```
CvCapture* cvCaptureFromFile( const char* filename );
```

filename Name of the video file.

The function `cvCaptureFromFile` allocates and initializes the CvCapture structure for reading the video stream from the specified file. Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect and how to prepare your video files.

After the allocated structure is not used any more it should be released by the ReleaseCapture function.

cvGetCaptureProperty

Gets video capturing properties.

```
double cvGetCaptureProperty( CvCapture* capture, int property_id );
```

capture video capturing structure.

property_id Property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file

CV_CAP_PROP_BRIGHTNESS Brightness of the image (only for cameras)

CV_CAP_PROP_CONTRAST Contrast of the image (only for cameras)

CV_CAP_PROP_SATURATION Saturation of the image (only for cameras)

CV_CAP_PROP_HUE Hue of the image (only for cameras)

The function `cvGetCaptureProperty` retrieves the specified property of the camera or video file.

---

cvGrabFrame

Grabs the frame from a camera or file.

```
int cvGrabFrame( CvCapture* capture );
```

capture video capturing structure.

The function `cvGrabFrame` grabs the frame from a camera or file. The grabbed frame is stored internally. The purpose of this function is to grab the frame *quickly* so that syncronization can occur if it has to read from several cameras simultaneously. The grabbed frames are not exposed because they may be stored in a compressed format (as defined by the camera/driver). To retrieve the grabbed frame, RetrieveFrame should be used.

---

cvQueryFrame

Grabs and returns a frame from a camera or file.

```
IplImage* cvQueryFrame( CvCapture* capture );
```

capture video capturing structure.

The function `cvQueryFrame` grabs a frame from a camera or video file, decompresses it and returns it. This function is just a combination of GrabFrame and RetrieveFrame , but in one call. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cvReleaseCapture

Releases the CvCapture structure.

```
void cvReleaseCapture( CvCapture** capture );
```

capture Pointer to video the capturing structure.

The function `cvReleaseCapture` releases the CvCapture structure allocated by CaptureFromFile or CaptureFromCAM .

cvRetrieveFrame

Gets the image grabbed with cvGrabFrame.

```
IplImage* cvRetrieveFrame( CvCapture* capture );
```

capture video capturing structure.

The function `cvRetrieveFrame` returns the pointer to the image grabbed with the GrabFrame function. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cvSetCaptureProperty

Sets video capturing properties.

```
int cvSetCaptureProperty(
            CvCapture* capture,
            int property_id,
            double value );
```

capture video capturing structure.

property_id property identifier. Can be one of the following:

CV_CAP_PROP_POS_MSEC Film current position in milliseconds or video capture timestamp

CV_CAP_PROP_POS_FRAMES 0-based index of the frame to be decoded/captured next

CV_CAP_PROP_POS_AVI_RATIO Relative position of the video file (0 - start of the film, 1 - end of the film)

CV_CAP_PROP_FRAME_WIDTH Width of the frames in the video stream

CV_CAP_PROP_FRAME_HEIGHT Height of the frames in the video stream

CV_CAP_PROP_FPS Frame rate

CV_CAP_PROP_FOURCC 4-character code of codec

CV_CAP_PROP_BRIGHTNESS Brightness of the image (only for cameras)

CV_CAP_PROP_CONTRAST Contrast of the image (only for cameras)

CV_CAP_PROP_SATURATION Saturation of the image (only for cameras)

CV_CAP_PROP_HUE Hue of the image (only for cameras)

value value of the property.

The function `cvSetCaptureProperty` sets the specified property of video capturing. Currently the function supports only video files: `CV_CAP_PROP_POS_MSEC, CV_CAP_PROP_POS_FRAMES, CV_CAP_PROP_POS_AVI_RATIO`.

NB This function currently does nothing when using the latest CVS download on linux with FFMPEG (the function contents are hidden if 0 is used and returned).

---

cvCreateVideoWriter

Creates the video file writer.

```
typedef struct CvVideoWriter CvVideoWriter; CvVideoWriter*
cvCreateVideoWriter(
           const char* filename,
           int fourcc,
           double fps,
           CvSize frame_size,
           int is_color=1 );
```

filename Name of the output video file.

fourcc 4-character code of codec used to compress the frames. For example, `CV_FOURCC('P','I','M','1')` is a MPEG-1 codec, `CV_FOURCC('M','J','P','G')` is a motion-jpeg codec etc. Under Win32 it is possible to pass -1 in order to choose compression method and additional compression parameters from dialog. Under Win32 if 0 is passed while using an avi filename it will create a video writer that creates an uncompressed avi file.

fps Framerate of the created video stream.

frame_size Size of the video frames.

is_color If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only).

The function `cvCreateVideoWriter` creates the video writer structure.

Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect.

---

cvReleaseVideoWriter

Releases the AVI writer.

```
void cvReleaseVideoWriter( CvVideoWriter** writer );
```

writer Pointer to the video file writer structure.

The function `cvReleaseVideoWriter` finishes writing to the video file and releases the structure.

---

cvWriteFrame

Writes a frame to a video file.

```
int cvWriteFrame( CvVideoWriter* writer, const IplImage* image );
```

writer Video writer structure

image The written frame

The function `cvWriteFrame` writes/appends one frame to a video file.

Chapter 5

ml. Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different seta of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class ‘CvStatModel‘ that all the other ML classes are derived from.

Part II

C++ API Reference

Chapter 6

Introduction

Starting from OpenCV 2.0 the new modern C++ interface has been introduced. It is crisp (less typing is needed to code the same thing), type-safe (no more `CvArr*` a.k.a. `void*`) and, in general, more convenient to use. Here is a short example of what it looks like:

```
//
// Simple retro-style photo effect done by adding noise to
// the luminance channel and reducing intensity of the chroma channels
//

// include standard OpenCV headers, same as before
#include "cv.h"
#include "highgui.h"

// all the new API is put into "cv" namespace. Export its content
using namespace cv;

// enable/disable use of mixed API in the code below.
#define DEMO_MIXED_API_USE 1

int main( int argc, char** argv )
{
    const char* imagename = argc > 1 ? argv[1] : "lena.jpg";
#if DEMO_MIXED_API_USE
    // Ptr<T> is safe ref-conting pointer class
    Ptr<IplImage> iplimg = cvLoadImage(imagename);

    // cv::Mat replaces the CvMat and IplImage, but it's easy to convert
    // between the old and the new data structures
    // (by default, only the header is converted and the data is shared)
    Mat img(iplimg);
```

```
#else
    // the newer cvLoadImage alternative with MATLAB-style name
    Mat img = imread(imagename);
#endif

    if( !img.data ) // check if the image has been loaded properly
        return -1;

    Mat img_yuv;
    // convert image to YUV color space.
    // The output image will be allocated automatically
    cvtColor(img, img_yuv, CV_BGR2YCrCb);

    // split the image into separate color planes
    vector<Mat> planes;
    split(img_yuv, planes);

    // another Mat constructor; allocates a matrix of the specified
        // size and type
    Mat noise(img.size(), CV_8U);

    // fills the matrix with normally distributed random values;
    // there is also randu() for uniformly distributed random numbers.
    // Scalar replaces CvScalar, Scalar::all() replaces cvScalarAll().
    randn(noise, Scalar::all(128), Scalar::all(20));

    // blur the noise a bit, kernel size is 3x3 and both sigma's
        // are set to 0.5
    GaussianBlur(noise, noise, Size(3, 3), 0.5, 0.5);

    const double brightness_gain = 0;
    const double contrast_gain = 1.7;
#if DEMO_MIXED_API_USE
    // it's easy to pass the new matrices to the functions that
    // only work with IplImage or CvMat:
    // step 1) - convert the headers, data will not be copied
    IplImage cv_planes_0 = planes[0], cv_noise = noise;
    // step 2) call the function; do not forget unary "&" to form pointers
    cvAddWeighted(&cv_planes_0, contrast_gain, &cv_noise, 1,
                 -128 + brightness_gain, &cv_planes_0);
#else
    addWeighted(planes[0], constrast_gain, noise, 1,
                -128 + brightness_gain, planes[0]);
#endif
    const double color_scale = 0.5;
```

```
    // Mat::convertTo() replaces cvConvertScale.
    // One must explicitly specify the output matrix type
    // (we keep it intact, i.e. pass planes[1].type())
    planes[1].convertTo(planes[1], planes[1].type(),
                        color_scale, 128*(1-color_scale));

    // alternative form of convertTo if we know the datatype
    // at compile time ("uchar" here).
    // This expression will not create any temporary arrays
    // and should be almost as fast as the above variant
    planes[2] = Mat_<uchar>(planes[2]*color_scale + 128*(1-color_scale));

    // Mat::mul replaces cvMul(). Again, no temporary arrays are
    // created in the case of simple expressions.
    planes[0] = planes[0].mul(planes[0], 1./255);

    // now merge the results back
    merge(planes, img_yuv);
    // and produce the output RGB image
    cvtColor(img_yuv, img, CV_YCrCb2BGR);

    // this is counterpart for cvNamedWindow
    namedWindow("image with grain", CV_WINDOW_AUTOSIZE);
#if DEMO_MIXED_API_USE
    // this is to demonstrate that img and iplimg really share the data -
    // the result of the above processing is stored to img and thus
        // in iplimg too.
    cvShowImage("image with grain", iplimg);
#else
    imshow("image with grain", img);
#endif
    waitKey();

    return 0;
    // all the memory will automatically be released
    // by vector<>, Mat and Ptr<> destructors.
}
```

Following a summary ”cheatsheet” below, the rest of the introduction will discuss the key features of the new interface in more detail.

6.1 C++ Cheatsheet

The section is just a summary ”cheatsheet” of common things you may want to do with cv::Mat:. The code snippets below all assume the correct namespace is used:

```
using namespace cv;
using namespace std;
```

Convert an IplImage or CvMat to an cv::Mat and a cv::Mat to an IplImage or CvMat:

```
// Assuming somewhere IplImage *iplimg; exists
// and has been allocated and cv::Mat Mimg has been defined
Mat imgMat(iplimg);  //Construct an Mat image "img" out of an IplImage
Mimg = iplimg;       //Or just set the header of pre existing cv::Mat
                     //Ming to iplimg's data (no copying is done)

//Convert to IplImage or CvMat, no data copying
IplImage ipl_img = img;
CvMat cvmat = img; // convert cv::Mat -> CvMat
```

A very simple way to operate on a rectanglular sub-region of an image (ROI – ”Region of Interest”):

```
//Make a rectangle
Rect roi(10, 20, 100, 50);
//Point a cv::Mat header at it (no allocation is done)
Mat image_roi = image(roi);
```

A bit advanced, but should you want efficiently to sample from a circular region in an image (below, instead of sampling, we just draw into a BGR image) :

```
// the function returns x boundary coordinates of
// the circle for each y. RxV[y1] = x1 means that
// when y=y1, -x1 <=x<=x1 is inside the circle
void getCircularROI(int R, vector < int > & RxV)
{
    RxV.resize(R+1);
    for( int y = 0; y <= R; y++ )
        RxV[y] = cvRound(sqrt((double)R*R - y*y));
}

// This draws a circle in the green channel
// (note the "[1]" for a BGR" image,
// blue and red channels are not modified),
// but is really an example of how to *sample* from a circular region.
void drawCircle(Mat &image, int R, Point center)
{
    vector<int> RxV;
```

```
    getCircularROI(R, RxV);

    Mat_<Vec3b>& img = (Mat_<Vec3b>&)image; //3 channel pointer to image
    for( int dy = -R; dy <= R; dy++ )
    {
        int Rx = RxV[abs(dy)];
        for( int dx = -Rx; dx <= Rx; dx++ )
            img(center.y+dy, center.x+dx)[1] = 255;
    }
}
```

6.2 Namespace cv and Function Naming

All the newly introduced classes and functions are placed into `cv` namespace. Therefore, to access this functionality from your code, use `cv::` specifier or "`using namespace cv;`" directive:

```
#include "cv.h"

...
cv::Mat H = cv::findHomography(points1, points2, cv::RANSAC, 5);
...
```

or

```
#include "cv.h"

using namespace cv;

...
Mat H = findHomography(points1, points2, RANSAC, 5 );
...
```

It is probable that some of the current or future OpenCV external names conflict with STL or other libraries, in this case use explicit namespace specifiers to resolve the name conflicts:

```
Mat a(100, 100, CV_32F);
randu(a, Scalar::all(1), Scalar::all(std::rand()%256+1));
cv::log(a, a);
a /= std::log(2.);
```

For the most of the C functions and structures from OpenCV 1.x you may find the direct counterparts in the new C++ interface. The name is usually formed by omitting `cv` or `Cv` prefix and turning the first letter to the low case (unless it's a own name, like Canny, Sobel etc). In case when there is no the new-style counterpart, it's possible to use the old functions with the new structures, as shown the first sample in the chapter.

6.3 Memory Management

When using the new interface, the most of memory deallocation and even memory allocation operations are done automatically when needed.

First of all, Mat , SparseMat and other classes have destructors that deallocate memory buffers occupied by the structures when needed.

Secondly, this "when needed" means that the destructors do not always deallocate the buffers, they take into account possible data sharing. That is, in a destructor the reference counter associated with the underlying data is decremented and the data is deallocated if and only if the reference counter becomes zero, that is, when no other structures refer to the same buffer. When such a structure containing a reference counter is copied, usually just the header is duplicated, while the underlying data is not; instead, the reference counter is incremented to memorize that there is another owner of the same data. Also, some structures, such as `Mat`, can refer to the user-allocated data. In this case the reference counter is `NULL` pointer and then no reference counting is done - the data is not deallocated by the destructors and should be deallocated manually by the user. We saw this scheme in the first example in the chapter:

```
// allocates IplImages and wraps it into shared pointer class.
Ptr<IplImage> iplimg = cvLoadImage(...);

// constructs Mat header for IplImage data;
// does not copy the data;
// the reference counter will be NULL
Mat img(iplimg);
...
// in the end of the block img destructor is called,
// which does not try to deallocate the data because
// of NULL pointer to the reference counter.
//
// Then Ptr<IplImage> destructor is called that decrements
// the reference counter and, as the counter becomes 0 in this case,
// the destructor calls cvReleaseImage().
```

The copying semantics was mentioned in the above paragraph, but deserves a dedicated discussion. By default, the new OpenCV structures implement shallow, so called O(1) (i.e. constant-time) assignment operations. It gives user possibility to pass quite big data structures to functions (though, e.g. passing `const Mat&` is still faster than passing `Mat`), return them (e.g. see the example with findHomography above), store them in OpenCV and STL containers etc. - and do all of this very efficiently. On the other hand, most of the new data structures provide `clone()` method that creates a full copy of an object. Here is the sample:

```
// create a big 8Mb matrix
Mat A(1000, 1000, CV_64F);
```

```
// create another header for the same matrix;
// this is instant operation, regardless of the matrix size.
Mat B = A;
// create another header for the 3-rd row of A; no data is copied either
Mat C = B.row(3);
// now create a separate copy of the matrix
Mat D = B.clone();
// copy the 5-th row of B to C, that is, copy the 5-th row of A
// to the 3-rd row of A.
B.row(5).copyTo(C);
// now let A and D share the data; after that the modified version
// of A is still referenced by B and C.
A = D;
// now make B an empty matrix (which references no memory buffers),
// but the modified version of A will still be referenced by C,
// despite that C is just a single row of the original A
B.release();

// finally, make a full copy of C. In result, the big modified
// matrix will be deallocated, since it's not referenced by anyone
C = C.clone();
```

Memory management of the new data structures is automatic and thus easy. If, however, your code uses IplImage , CvMat or other C data structures a lot, memory management can still be automated without immediate migration to Mat by using the already mentioned template class Ptr , similar to `shared_ptr` from Boost and C++ TR1. It wraps a pointer to an arbitrary object, provides transparent access to all the object fields and associates a reference counter with it. Instance of the class can be passed to any function that expects the original pointer. For correct deallocation of the object, you should specialize `Ptr<T>::delete_obj()` method. Such specialized methods already exist for the classical OpenCV structures, e.g.:

```
// cxoperations.hpp:
...
template<> inline Ptr<IplImage>::delete_obj() {
    cvReleaseImage(&obj);
}
...
```

See Ptr description for more details and other usage scenarios.

6.4 Memory Management Part II. Automatic Data Allocation

With the new interface not only explicit memory deallocation is not needed anymore, but the memory allocation is often done automatically too. That was demonstrated in the example in the be-

ginning of the chapter when `cvtColor` was called, and here are some more details.

Mat and other array classes provide method `create` that allocates a new buffer for array data if and only if the currently allocated array is not of the required size and type. If a new buffer is needed, the previously allocated buffer is released (by engaging all the reference counting mechanism described in the previous section). Now, since it is very quick to check whether the needed memory buffer is already allocated, most new OpenCV functions that have arrays as output parameters call the `create` method and this way the *automatic data allocation* concept is implemented. Here is the example:

```
#include "cv.h"
#include "highgui.h"

using namespace cv;

int main(int, char**)
{
    VideoCapture cap(0);
    if(!cap.isOpened()) return -1;

    Mat edges;
    namedWindow("edges",1);
    for(;;)
    {
        Mat frame;
        cap >> frame;
        cvtColor(frame, edges, CV_BGR2GRAY);
        GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
        Canny(edges, edges, 0, 30, 3);
        imshow("edges", edges);
        if(waitKey(30) >= 0) break;
    }
    return 0;
}
```

The matrix `edges` is allocated during the first frame processing and unless the resolution will suddenly change, the same buffer will be reused for every next frame's edge map.

In many cases the output array type and size can be inferenced from the input arrays' respective characteristics, but not always. In these rare cases the corresponding functions take separate input parameters that specify the data type and/or size of the output arrays, like resize . Anyway, a vast majority of the new-style array processing functions call `create` for each of the output array, with just a few exceptions like `mixChannels`, `RNG::fill` and some others.

Note that this output array allocation semantic is only implemented in the new functions. If you want to pass the new structures to some old OpenCV function, you should first allocate the output arrays using `create` method, then make `CvMat` or `IplImage` headers and after that call

the function.

6.5 Algebraic Operations

Just like in v1.x, OpenCV 2.x provides some basic functions operating on matrices, like `add`, `subtract`, `gemm` etc. In addition, it introduces overloaded operators that give the user a convenient algebraic notation, which is nearly as fast as using the functions directly. For example, here is how the least squares problem $Ax = b$ can be solved using normal equations:

```
Mat x = (A.t()*A).inv()*(A.t()*b);
```

The complete list of overloaded operators can be found in Matrix Expressions .

6.6 Fast Element Access

Historically, OpenCV provided many different ways to access image and matrix elements, and none of them was both fast and convenient. With the new data structures, OpenCV 2.x introduces a few more alternatives, hopefully more convenient than before. For detailed description of the operations, please, check Mat and Mat_ description. Here is part of the retro-photo-styling example rewritten (in simplified form) using the element access operations:

```
...
// split the image into separate color planes
vector<Mat> planes;
split(img_yuv, planes);

// method 1. process Y plane using an iterator
MatIterator_<uchar> it = planes[0].begin<uchar>(),
                    it_end = planes[0].end<uchar>();
for(; it != it_end; ++it)
{
    double v = *it*1.7 + rand()%21-10;
    *it = saturate_cast<uchar>(v*v/255.);
}

// method 2. process the first chroma plane using pre-stored row pointer.
// method 3. process the second chroma plane using
             individual element access operations
for( int y = 0; y < img_yuv.rows; y++ )
{
    uchar* Uptr = planes[1].ptr<uchar>(y);
    for( int x = 0; x < img_yuv.cols; x++ )
    {
        Uptr[x] = saturate_cast<uchar>((Uptr[x]-128)/2 + 128);
```

```
        uchar& Vxy = planes[2].at<uchar>(y, x);
        Vxy = saturate_cast<uchar>((Vxy-128)/2 + 128);
    }
}

merge(planes, img_yuv);
...
```

6.7 Saturation Arithmetics

In the above sample you may have noticed saturate_cast operator, and that's how all the pixel processing is done in OpenCV. When a result of image operation is 8-bit image with pixel values ranging from 0 to 255, each output pixel value is clipped to this available range:

$$I(x, y) = \min(\max(value, 0), 255)$$

and the similar rules are applied to 8-bit signed and 16-bit signed and unsigned types. This "saturation" semantics (different from usual C language "wrapping" semantics, where lowest bits are taken, is implemented in every image processing function, from the simple `cv::add` to `cv::cvtColor`, `cv::resize`, `cv::filter2D` etc. It is not a new feature of OpenCV v2.x, it was there from very beginning. In the new version this special saturate_cast template operator is introduced to simplify implementation of this semantic in your own functions.

6.8 Error handling

The modern error handling mechanism in OpenCV uses exceptions, as opposite to the manual stack unrolling used in previous versions. When OpenCV is built in DEBUG configuration, the error handler provokes memory access violation, so that the full call stack and context can be analyzed with debugger.

6.9 Threading and Reenterability

OpenCV uses OpenMP to run some time-consuming operations in parallel. Threading can be explicitly controlled by setNumThreads function. Also, functions and "const" methods of the classes are generally re-enterable, that is, they can be called from different threads asynchronously.

Chapter 7

cxcore. The Core Functionality

7.1 Basic Structures

DataType

Template ”traits” class for other OpenCV primitive data types

```
template<typename _Tp> class DataType
{
    // value_type is always a synonym for _Tp.
    typedef _Tp value_type;

    // intermediate type used for operations on _Tp.
    // it is int for uchar, signed char, unsigned short, signed short and int,
    // float for float, double for double, ...
    typedef <...> work_type;
    // in the case of multi-channel data it is the data type of each channel
    typedef <...> channel_type;
    enum
    {
        // CV_8U ... CV_64F
        depth = DataDepth<channel_type>::value,
        // 1 ...
        channels = <...>,
        // ’1u’, ’4i’, ’3f’, ’2d’ etc.
        fmt=<...>,
        // CV_8UC3, CV_32FC2 ...
        type = CV_MAKETYPE(depth, channels)
    };
};
```

The template class `DataType` is descriptive class for OpenCV primitive data types and other types that comply with the following definition. A primitive OpenCV data type is one of `unsigned char, bool (~unsigned char), signed char, unsigned short, signed short, int, float, double` or a tuple of values of one of these types, where all the values in the tuple have the same type. If you are familiar with OpenCV CvMat 's type notation, CV_8U ... CV_32FC3, CV_64FC2 etc., then a primitive type can be defined as a type for which you can give a unique identifier in a form `CV\_<bit-depth>{U|S|F}C<number_of_channels>`. A universal OpenCV structure able to store a single instance of such primitive data type is Vec . Multiple instances of such a type can be stored to a `std::vector`, `Mat`, `Mat_`, `MatND`, `MatND_`, `SparseMat`, `SparseMat_` or any other container that is able to store Vec instances.

The class `DataType` is basically used to provide some description of such primitive data types without adding any fields or methods to the corresponding classes (and it is actually impossible to add anything to primitive C/C++ data types). This technique is known in C++ as class traits. It's not `DataType` itself that is used, but its specialized versions, such as:

```
template<> class DataType<uchar>
{
    typedef uchar value_type;
    typedef int work_type;
    typedef uchar channel_type;
    enum { channel_type = CV_8U, channels = 1, fmt='u', type = CV_8U };
};
...
template<typename _Tp> DataType<std::complex<_Tp> >
{
    typedef std::complex<_Tp> value_type;
    typedef std::complex<_Tp> work_type;
    typedef _Tp channel_type;
    // DataDepth is another helper trait class
    enum { depth = DataDepth<_Tp>::value, channels=2,
        fmt=(channels-1)*256+DataDepth<_Tp>::fmt,
        type=CV_MAKETYPE(depth, channels) };
};
...
```

The main purpose of the classes is to convert compile-time type information to OpenCV-compatible data type identifier, for example:

```
// allocates 30x40 floating-point matrix
Mat A(30, 40, DataType<float>::type);

Mat B = Mat_<std::complex<double> >(3, 3);
// the statement below will print 6, 2 /* i.e. depth == CV_64F, channels == 2 */
cout << B.depth() << ", " << B.channels() << endl;
```

that is, such traits are used to tell OpenCV which data type you are working with, even if such a type is not native to OpenCV (the matrix `B` intialization above compiles because OpenCV defines the proper specialized template class `DataType<complex<_Tp> >`). Also, this mechanism is useful (and used in OpenCV this way) for generic algorithms implementations.

Point_

Template class for 2D points

```
template<typename _Tp> class Point_
{
public:
    typedef _Tp value_type;

    Point_();
    Point_(_Tp _x, _Tp _y);
    Point_(const Point_& pt);
    Point_(const CvPoint& pt);
    Point_(const CvPoint2D32f& pt);
    Point_(const Size_<_Tp>& sz);
    Point_(const Vec<_Tp, 2>& v);
    Point_& operator = (const Point_& pt);
    template<typename _Tp2> operator Point_<_Tp2>() const;
    operator CvPoint() const;
    operator CvPoint2D32f() const;
    operator Vec<_Tp, 2>() const;

    // computes dot-product (this->x*pt.x + this->y*pt.y)
    _Tp dot(const Point_& pt) const;
    // computes dot-product using double-precision arithmetics
    double ddot(const Point_& pt) const;
    // returns true if the point is inside the rectangle "r".
    bool inside(const Rect_<_Tp>& r) const;

    _Tp x, y;
};
```

The class represents a 2D point, specified by its coordinates x and y. Instance of the class is interchangeable with C structures `CvPoint` and `CvPoint2D32f`. There is also cast operator to convert point coordinates to the specified type. The conversion from floating-point coordinates to integer coordinates is done by rounding; in general case the conversion uses saturate_cast operation on each of the coordinates. Besides the class members listed in the declaration above, the following operations on points are implemented:

```
    pt1 = pt2 + pt3;
```

```
    pt1 = pt2 - pt3;
    pt1 = pt2 * a;
    pt1 = a * pt2;
    pt1 += pt2;
    pt1 -= pt2;
    pt1 *= a;
    double value = norm(pt); // L2 norm
    pt1 == pt2;
    pt1 != pt2;
```

For user convenience, the following type aliases are defined:

```
typedef Point_<int> Point2i;
typedef Point2i Point;
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;
```

Here is a short example:

```
Point2f a(0.3f, 0.f), b(0.f, 0.4f);
Point pt = (a + b)*10.f;
cout << pt.x << ", " << pt.y << endl;
```

Point3_

Template class for 3D points

```
template<typename _Tp> class Point3_
{
public:
    typedef _Tp value_type;

    Point3_();
    Point3_(_Tp _x, _Tp _y, _Tp _z);
    Point3_(const Point3_& pt);
    explicit Point3_(const Point_<_Tp>& pt);
    Point3_(const CvPoint3D32f& pt);
    Point3_(const Vec<_Tp, 3>& v);
    Point3_& operator = (const Point3_& pt);
    template<typename _Tp2> operator Point3_<_Tp2>() const;
    operator CvPoint3D32f() const;
    operator Vec<_Tp, 3>() const;

    _Tp dot(const Point3_& pt) const;
    double ddot(const Point3_& pt) const;
```

```
    _Tp x, y, z;
};
```

The class represents a 3D point, specified by its coordinates x, y and z. Instance of the class is interchangeable with C structure `CvPoint2D32f`. Similarly to `Point_`, the 3D points' coordinates can be converted to another type, and the vector arithmetic and comparison operations are also supported.

The following type aliases are available:

```
typedef Point3_<int> Point3i;
typedef Point3_<float> Point3f;
typedef Point3_<double> Point3d;
```

Size_

Template class for specfying image or rectangle size.

```
template<typename _Tp> class Size_
{
public:
    typedef _Tp value_type;

    Size_();
    Size_(_Tp _width, _Tp _height);
    Size_(const Size_& sz);
    Size_(const CvSize& sz);
    Size_(const CvSize2D32f& sz);
    Size_(const Point_<_Tp>& pt);
    Size_& operator = (const Size_& sz);
    _Tp area() const;

    operator Size_<int>() const;
    operator Size_<float>() const;
    operator Size_<double>() const;
    operator CvSize() const;
    operator CvSize2D32f() const;

    _Tp width, height;
};
```

The class `Size_` is similar to `Point_`, except that the two members are called `width` and `height` instead of `x` and `y`. The structure can be converted to and from the old OpenCV structures CvSize and CvSize2D32f . The same set of arithmetic and comparison operations as for `Point_` is available.

OpenCV defines the following type aliases:

```
typedef Size_<int> Size2i;
typedef Size2i Size;
typedef Size_<float> Size2f;
```

Rect_

Template class for 2D rectangles

```
template<typename _Tp> class Rect_
{
public:
    typedef _Tp value_type;

    Rect_();
    Rect_(_Tp _x, _Tp _y, _Tp _width, _Tp _height);
    Rect_(const Rect_& r);
    Rect_(const CvRect& r);
    // (x, y) <- org, (width, height) <- sz
    Rect_(const Point_<_Tp>& org, const Size_<_Tp>& sz);
    // (x, y) <- min(pt1, pt2), (width, height) <- max(pt1, pt2) - (x, y)
    Rect_(const Point_<_Tp>& pt1, const Point_<_Tp>& pt2);
    Rect_& operator = ( const Rect_& r );
    // returns Point_<_Tp>(x, y)
    Point_<_Tp> tl() const;
    // returns Point_<_Tp>(x+width, y+height)
    Point_<_Tp> br() const;

    // returns Size_<_Tp>(width, height)
    Size_<_Tp> size() const;
    // returns width*height
    _Tp area() const;

    operator Rect_<int>() const;
    operator Rect_<float>() const;
    operator Rect_<double>() const;
    operator CvRect() const;

    // x <= pt.x && pt.x < x + width &&
    // y <= pt.y && pt.y < y + height ? true : false
    bool contains(const Point_<_Tp>& pt) const;

    _Tp x, y, width, height;
};
```

The rectangle is described by the coordinates of the top-left corner (which is the default interpretation of `Rect_::x` and `Rect_::y` in OpenCV; though, in your algorithms you may count `x` and `y` from the bottom-left corner), the rectangle width and height.

Another assumption OpenCV usually makes is that the top and left boundary of the rectangle are inclusive, while the right and bottom boundaries are not, for example, the method `Rect_::contains` returns true if

$$x \leq pt.x < x + width,$$
$$y \leq pt.y < y + height$$

And virtually every loop over an image ROI in OpenCV (where ROI is specified by `Rect_<int>`) is implemented as:

```
for(int y = roi.y; y < roi.y + rect.height; y++)
    for(int x = roi.x; x < roi.x + rect.width; x++)
    {
        // ...
    }
```

In addition to the class members, the following operations on rectangles are implemented:

- `rect = rect ± point` (shifting rectangle by a certain offset)
- `rect = rect ± size` (expanding or shrinking rectangle by a certain amount)
- `rect += point, rect -= point, rect += size, rect -= size` (augmenting operations)
- `rect = rect1 & rect2` (rectangle intersection)
- `rect = rect1 | rect2` (minimum area rectangle containing `rect2` and `rect3`)
- `rect &= rect1, rect |= rect1` (and the corresponding augmenting operations)
- `rect == rect1, rect != rect1` (rectangle comparison)

Example. Here is how the partial ordering on rectangles can be established (rect1 $\subseteq$ rect2):

```
template<typename _Tp> inline bool
operator <= (const Rect_<_Tp>& r1, const Rect_<_Tp>& r2)
{
    return (r1 & r2) == r1;
}
```

For user convenience, the following type alias is available:

```
typedef Rect_<int> Rect;
```

RotatedRect

Possibly rotated rectangle

```
class RotatedRect
{
public:
    // constructors
    RotatedRect();
    RotatedRect(const Point2f& _center, const Size2f& _size, float _angle);
    RotatedRect(const CvBox2D& box);

    // returns minimal up-right rectangle that contains the rotated rectangle
    Rect boundingRect() const;
    // backward conversion to CvBox2D
    operator CvBox2D() const;

    // mass center of the rectangle
    Point2f center;
    // size
    Size2f size;
    // rotation angle in degrees
    float angle;
};
```

The class `RotatedRect` replaces the old CvBox2D and fully compatible with it.

TermCriteria

Termination criteria for iterative algorithms

```
class TermCriteria
{
public:
    enum { COUNT=1, MAX_ITER=COUNT, EPS=2 };

    // constructors
    TermCriteria();
    // type can be MAX_ITER, EPS or MAX_ITER+EPS.
    // type = MAX_ITER means that only the number of iterations does matter;
    // type = EPS means that only the required precision (epsilon) does matter
    //    (though, most algorithms put some limit on the number of iterations anyway)
    // type = MAX_ITER + EPS means that algorithm stops when
    // either the specified number of iterations is made,
    // or when the specified accuracy is achieved - whatever happens first.
```

```
    TermCriteria(int _type, int _maxCount, double _epsilon);
    TermCriteria(const CvTermCriteria& criteria);
    operator CvTermCriteria() const;

    int type;
    int maxCount;
    double epsilon;
};
```

The class `TermCriteria` replaces the old CvTermCriteria and fully compatible with it.

Vec

Template class for short numerical vectors

```
template<typename _Tp, int cn> class Vec
{
public:
    typedef _Tp value_type;
    enum { depth = DataDepth<_Tp>::value, channels = cn,
           type = CV_MAKETYPE(depth, channels) };

    // default constructor: all elements are set to 0
    Vec();
    // constructors taking up to 10 first elements as parameters
    Vec(_Tp v0);
    Vec(_Tp v0, _Tp v1);
    Vec(_Tp v0, _Tp v1, _Tp v2);
    ...
    Vec(_Tp v0, _Tp v1, _Tp v2, _Tp v3, _Tp v4,
        _Tp v5, _Tp v6, _Tp v7, _Tp v8, _Tp v9);
    Vec(const Vec<_Tp, cn>& v);
    // constructs vector with all the components set to alpha.
    static Vec all(_Tp alpha);

    // two variants of dot-product
    _Tp dot(const Vec& v) const;
    double ddot(const Vec& v) const;

    // cross-product; valid only when cn == 3.
    Vec cross(const Vec& v) const;

    // element type conversion
    template<typename T2> operator Vec<T2, cn>() const;
```

```
    // conversion to/from CvScalar (valid only when cn==4)
    operator CvScalar() const;

    // element access
    _Tp operator [](int i) const;
    _Tp& operator[](int i);

    _Tp val[cn];
};
```

The class is the most universal representation of short numerical vectors or tuples. It is possible to convert `Vec<T,2>` to/from `Point_`, `Vec<T,3>` to/from `Point3_`, and `Vec<T,4>` to CvScalar . The elements of `Vec` are accessed using `operator[]`. All the expected vector operations are implemented too:

- `v1 = v2 ± v3`, `v1 = v2 * α`, `v1 = α * v2` (plus the corresponding augmenting operations; note that these operations apply saturate_cast.3C.3E to the each computed vector component)
- `v1 == v2`, `v1 != v2`
- `double n = norm(v1); // ` L_2`-norm`

For user convenience, the following type aliases are introduced:

```
typedef Vec<uchar, 2> Vec2b;
typedef Vec<uchar, 3> Vec3b;
typedef Vec<uchar, 4> Vec4b;

typedef Vec<short, 2> Vec2s;
typedef Vec<short, 3> Vec3s;
typedef Vec<short, 4> Vec4s;

typedef Vec<int, 2> Vec2i;
typedef Vec<int, 3> Vec3i;
typedef Vec<int, 4> Vec4i;

typedef Vec<float, 2> Vec2f;
typedef Vec<float, 3> Vec3f;
typedef Vec<float, 4> Vec4f;
typedef Vec<float, 6> Vec6f;

typedef Vec<double, 2> Vec2d;
typedef Vec<double, 3> Vec3d;
typedef Vec<double, 4> Vec4d;
typedef Vec<double, 6> Vec6d;
```

The class `Vec` can be used for declaring various numerical objects, e.g. `Vec<double, 9>` can be used to store a 3x3 double-precision matrix. It is also very useful for declaring and processing multi-channel arrays, see `Mat_` description.

Scalar_

4-element vector

```
template<typename _Tp> class Scalar_ : public Vec<_Tp, 4>
{
public:
    Scalar_();
    Scalar_(_Tp v0, _Tp v1, _Tp v2=0, _Tp v3=0);
    Scalar_(const CvScalar& s);
    Scalar_(_Tp v0);
    static Scalar_<_Tp> all(_Tp v0);
    operator CvScalar() const;

    template<typename T2> operator Scalar_<T2>() const;

    Scalar_<_Tp> mul(const Scalar_<_Tp>& t, double scale=1 ) const;
    template<typename T2> void convertTo(T2* buf, int channels, int unroll_to=0) const;
};

typedef Scalar_<double> Scalar;
```

The template class `Scalar_` and it's double-precision instantiation `Scalar` represent 4-element vector. Being derived from `Vec<_Tp, 4>`, they can be used as typical 4-element vectors, but in addition they can be converted to/from `CvScalar`. The type `Scalar` is widely used in OpenCV for passing pixel values and it is a drop-in replacement for CvScalar that was used for the same purpose in the earlier versions of OpenCV.

Range

Specifies a continuous subsequence (a.k.a. slice) of a sequence.

```
class Range
{
public:
    Range();
    Range(int _start, int _end);
    Range(const CvSlice& slice);
    int size() const;
    bool empty() const;
```

```
    static Range all();
    operator CvSlice() const;

    int start, end;
};
```

The class is used to specify a row or column span in a matrix (Mat), and for many other purposes. `Range(a,b)` is basically the same as `a:b` in Matlab or `a..b` in Python. As in Python, `start` is inclusive left boundary of the range, and `end` is exclusive right boundary of the range. Such a half-opened interval is usually denoted as $[start, end)$.

The static method `Range::all()` returns some special variable that means "the whole sequence" or "the whole range", just like ":" in Matlab or "..." in Python. All the methods and functions in OpenCV that take `Range` support this special `Range::all()` value, but of course, in the case of your own custom processing you will probably have to check and handle it explicitly:

```
void my_function(..., const Range& r, ....)
{
    if(r == Range::all()) {
        // process all the data
    }
    else {
        // process [r.start, r.end)
    }
}
```

Ptr

A template class for smart reference-counting pointers

```
template<typename _Tp> class Ptr
{
public:
    // default constructor
    Ptr();
    // constructor that wraps the object pointer
    Ptr(_Tp* _obj);
    // destructor: calls release()
    ~Ptr();
    // copy constructor; increments ptr's reference counter
    Ptr(const Ptr& ptr);
    // assignment operator; decrements own reference counter
    // (with release()) and increments ptr's reference counter
    Ptr& operator = (const Ptr& ptr);
    // increments reference counter
```

```
    void addref();
    // decrements reference counter; when it becomes 0,
    // delete_obj() is called
    void release();
    // user-specified custom object deletion operation.
    // by default, "delete obj;" is called
    void delete_obj();
    // returns true if obj == 0;
    bool empty() const;

    // provide access to the object fields and methods
    _Tp* operator -> ();
    const _Tp* operator -> () const;

    // return the underlying object pointer;
    // thanks to the methods, the Ptr<_Tp> can be
    // used instead of _Tp*
    operator _Tp* ();
    operator const _Tp*() const;
protected:
    // the incapsulated object pointer
    _Tp* obj;
    // the associated reference counter
    int* refcount;
};
```

The class `Ptr<_Tp>` is a template class that wraps pointers of the corresponding type. It is similar to `shared_ptr` that is a part of Boost library (http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/shared_ptr.htm) and also a part of the C++0x standard.

By using this class you can get the following capabilities:

- default constructor, copy constructor and assignment operator for an arbitrary C++ class or a C structure. For some objects, like files, windows, mutexes, sockets etc, copy constructor or assignment operator are difficult to define. For some other objects, like complex classifiers in OpenCV, copy constructors are absent and not easy to implement. Finally, some of complex OpenCV and your own data structures may have been written in C. However, copy constructors and default constructors can simplify programming a lot; besides, they are often required (e.g. by STL containers). By wrapping a pointer to such a complex object `TObj` to `Ptr<TObj>` you will automatically get all of the necessary constructors and the assignment operator.
- all the above-mentioned operations running very fast, regardless of the data size, i.e. as "O(1)" operations. Indeed, while some structures, like `std::vector` provide a copy constructor and an assignment operator, the operations may take considerable time if the data

structures are big. But if the structures are put into `Ptr<>`, the overhead becomes small and independent of the data size.

- automatic destruction, even for C structures. See the example below with `FILE*`.
- heterogeneous collections of objects. The standard STL and most other C++ and OpenCV containers can only store objects of the same type and the same size. The classical solution to store objects of different types in the same container is to store pointers to the base class `base_class_t*` instead, but when you loose the automatic memory management. Again, by using `Ptr<base_class_t>()` instead of the raw pointers, you can solve the problem.

The class `Ptr` treats the wrapped object as a black box, the reference counter is allocated and managed separately. The only thing the pointer class needs to know about the object is how to deallocate it. This knowledge is incapsulated in `Ptr::delete_obj()` method, which is called when the reference counter becomes 0. If the object is a C++ class instance, no additional coding is needed, because the default implementation of this method calls `delete obj;`. However, if the object is deallocated in a different way, then the specialized method should be created. For example, if you want to wrap `FILE`, the `delete_obj` may be implemented as following:

```
template<> inline void Ptr<FILE>::delete_obj()
{
    fclose(obj); // no need to clear the pointer afterwards,
                 // it is done externally.
}
...

// now use it:
Ptr<FILE> f(fopen("myfile.txt", "r"));
if(f.empty())
    throw ...;
fprintf(f, ....);
...
// the file will be closed automatically by the Ptr<FILE> destructor.
```

Note: The reference increment/decrement operations are implemented as atomic operations, and therefore it is normally safe to use the classes in multi-threaded applications. The same is true for Mat and other C++ OpenCV classes that operate on the reference counters.

Mat

OpenCV C++ matrix class.

```
class Mat
{
```

```
public:
    // constructors
    Mat();
    // constructs matrix of the specified size and type
    // (_type is CV_8UC1, CV_64FC3, CV_32SC(12) etc.)
    Mat(int _rows, int _cols, int _type);
    // constucts matrix and fills it with the specified value _s.
    Mat(int _rows, int _cols, int _type, const Scalar& _s);
    Mat(Size _size, int _type);
    // copy constructor
    Mat(const Mat& m);
    // constructor for matrix headers pointing to user-allocated data
    Mat(int _rows, int _cols, int _type, void* _data, size_t _step=AUTO_STEP);
    Mat(Size _size, int _type, void* _data, size_t _step=AUTO_STEP);
    // creates a matrix header for a part of the bigger matrix
    Mat(const Mat& m, const Range& rowRange, const Range& colRange);
    Mat(const Mat& m, const Rect& roi);
    // converts old-style CvMat to the new matrix; the data is not copied by default
    Mat(const CvMat* m, bool copyData=false);
    // converts old-style IplImage to the new matrix; the data is not copied by default
    Mat(const IplImage* img, bool copyData=false);
    // builds matrix from std::vector with or without copying the data
    template<typename _Tp> Mat(const vector<_Tp>& vec, bool copyData=false);
    // helper constructor to compile matrix expressions
    Mat(const MatExpr_Base& expr);
    // destructor - calls release()
    ~Mat();
    // assignment operators
    Mat& operator = (const Mat& m);
    Mat& operator = (const MatExpr_Base& expr);

    ...
    // returns a new matrix header for the specified row
    Mat row(int y) const;
    // returns a new matrix header for the specified column
    Mat col(int x) const;
    // ... for the specified row span
    Mat rowRange(int startrow, int endrow) const;
    Mat rowRange(const Range& r) const;
    // ... for the specified column span
    Mat colRange(int startcol, int endcol) const;
    Mat colRange(const Range& r) const;
    // ... for the specified diagonal
    // (d=0 - the main diagonal,
    //  >0 - a diagonal from the lower half,
```

```
    //  <0 - a diagonal from the upper half)
    Mat diag(int d=0) const;
    // constructs a square diagonal matrix which main diagonal is vector "d"
    static Mat diag(const Mat& d);

    // returns deep copy of the matrix, i.e. the data is copied
    Mat clone() const;
    // copies the matrix content to "m".
    // It calls m.create(this->size(), this->type()).
    void copyTo( Mat& m ) const;
    // copies those matrix elements to "m" that are marked with non-zero mask elements.
    void copyTo( Mat& m, const Mat& mask ) const;
    // converts matrix to another datatype with optional scalng. See cvConvertScale.
    void convertTo( Mat& m, int rtype, double alpha=1, double beta=0 ) const;

    ...
    // sets every matrix element to s
    Mat& operator = (const Scalar& s);
    // sets some of the matrix elements to s, according to the mask
    Mat& setTo(const Scalar& s, const Mat& mask=Mat());
    // creates alternative matrix header for the same data, with different
    // number of channels and/or different number of rows. see cvReshape.
    Mat reshape(int _cn, int _rows=0) const;

    // matrix transposition by means of matrix expressions
    MatExpr_<...> t() const;
    // matrix inversion by means of matrix expressions
    MatExpr_<...> inv(int method=DECOMP_LU) const;
    // per-element matrix multiplication by means of matrix expressions
    MatExpr_<...> mul(const Mat& m, double scale=1) const;
    MatExpr_<...> mul(const MatExpr_<...>& m, double scale=1) const;

    // computes cross-product of 2 3D vectors
    Mat cross(const Mat& m) const;
    // computes dot-product
    double dot(const Mat& m) const;

    // Matlab-style matrix initialization. see the description
    static MatExpr_Initializer zeros(int rows, int cols, int type);
    static MatExpr_Initializer zeros(Size size, int type);
    static MatExpr_Initializer ones(int rows, int cols, int type);
    static MatExpr_Initializer ones(Size size, int type);
    static MatExpr_Initializer eye(int rows, int cols, int type);
    static MatExpr_Initializer eye(Size size, int type);
```

```
    // allocates new matrix data unless the matrix already has specified size and type.
    // previous data is unreferenced if needed.
    void create(int _rows, int _cols, int _type);
    void create(Size _size, int _type);
    // increases the reference counter; use with care to avoid memleaks
    void addref();
    // decreases reference counter;
    // deallocate the data when reference counter reaches 0.
    void release();

    // locates matrix header within a parent matrix. See below
    void locateROI( Size& wholeSize, Point& ofs ) const;
    // moves/resizes the current matrix ROI inside the parent matrix.
    Mat& adjustROI( int dtop, int dbottom, int dleft, int dright );
    // extracts a rectangular sub-matrix
    // (this is a generalized form of row, rowRange etc.)
    Mat operator()( Range rowRange, Range colRange ) const;
    Mat operator()( const Rect& roi ) const;

    // converts header to CvMat; no data is copied
    operator CvMat() const;
    // converts header to IplImage; no data is copied
    operator IplImage() const;

    // returns true iff the matrix data is continuous
    // (i.e. when there are no gaps between successive rows).
    // similar to CV_IS_MAT_CONT(cvmat->type)
    bool isContinuous() const;
    // returns element size in bytes,
    // similar to CV_ELEM_SIZE(cvmat->type)
    size_t elemSize() const;
    // returns the size of element channel in bytes.
    size_t elemSize1() const;
    // returns element type, similar to CV_MAT_TYPE(cvmat->type)
    int type() const;
    // returns element type, similar to CV_MAT_DEPTH(cvmat->type)
    int depth() const;
    // returns element type, similar to CV_MAT_CN(cvmat->type)
    int channels() const;
    // returns step/elemSize1()
    size_t step1() const;
    // returns matrix size:
    // width == number of columns, height == number of rows
    Size size() const;
    // returns true if matrix data is NULL
```

```
    bool empty() const;

    // returns pointer to y-th row
    uchar* ptr(int y=0);
    const uchar* ptr(int y=0) const;

    // template version of the above method
    template<typename _Tp> _Tp* ptr(int y=0);
    template<typename _Tp> const _Tp* ptr(int y=0) const;

    // template methods for read-write or read-only element access.
    // note that _Tp must match the actual matrix type -
    // the functions do not do any on-fly type conversion
    template<typename _Tp> _Tp& at(int y, int x);
    template<typename _Tp> _Tp& at(Point pt);
    template<typename _Tp> const _Tp& at(int y, int x) const;
    template<typename _Tp> const _Tp& at(Point pt) const;

    // template methods for iteration over matrix elements.
    // the iterators take care of skipping gaps in the end of rows (if any)
    template<typename _Tp> MatIterator_<_Tp> begin();
    template<typename _Tp> MatIterator_<_Tp> end();
    template<typename _Tp> MatConstIterator_<_Tp> begin() const;
    template<typename _Tp> MatConstIterator_<_Tp> end() const;

    enum { MAGIC_VAL=0x42FF0000, AUTO_STEP=0, CONTINUOUS_FLAG=CV_MAT_CONT_FLAG };

    // includes several bit-fields:
    //  * the magic signature
    //  * continuity flag
    //  * depth
    //  * number of channels
    int flags;
    // the number of rows and columns
    int rows, cols;
    // a distance between successive rows in bytes; includes the gap if any
    size_t step;
    // pointer to the data
    uchar* data;

    // pointer to the reference counter;
    // when matrix points to user-allocated data, the pointer is NULL
    int* refcount;

    // helper fields used in locateROI and adjustROI
```

```
    uchar* datastart;
    uchar* dataend;
};
```

The class `Mat` represents a 2D numerical array that can act as a matrix (and further it's referred to as a matrix), image, optical flow map etc. It is very similar to CvMat type from earlier versions of OpenCV, and similarly to `CvMat`, the matrix can be multi-channel, but it also fully supports ROI mechanism, just like IplImage .

There are many different ways to create `Mat` object. Here are the some popular ones:

- using `create(nrows, ncols, type)` method or the similar constructor `Mat(nrows, ncols, type[, fill_value])` constructor. A new matrix of the specified size and specifed type will be allocated. `type` has the same meaning as in cv::cvCreateMat method, e.g. `CV_8UC1` means 8-bit single-channel matrix, `CV_32FC2` means 2-channel (i.e. complex) floating-point matrix etc:

  ```
  // make 7x7 complex matrix filled with 1+3j.
  cv::Mat M(7,7,CV_32FC2,Scalar(1,3));
  // and now turn M to 100x60 15-channel 8-bit matrix.
  // The old content will be deallocated
  M.create(100,60,CV_8UC(15));
  ```

  As noted in the introduction of this chapter, `create()` will only allocate a new matrix when the current matrix dimensionality or type are different from the specified.

- by using a copy constructor or assignment operator, where on the right side it can be a matrix or expression, see below. Again, as noted in the introduction, matrix assignment is O(1) operation because it only copies the header and increases the reference counter. `Mat::clone()` method can be used to get a full (a.k.a. deep) copy of the matrix when you need it.

- by constructing a header for a part of another matrix. It can be a single row, single column, several rows, several columns, rectangular region in the matrix (called a minor in algebra) or a diagonal. Such operations are also O(1), because the new header will reference the same data. You can actually modify a part of the matrix using this feature, e.g.

  ```
  // add 5-th row, multiplied by 3 to the 3rd row
  M.row(3) = M.row(3) + M.row(5)*3;

  // now copy 7-th column to the 1-st column
  // M.col(1) = M.col(7); // this will not work
  Mat M1 = M.col(1);
  M.col(7).copyTo(M1);

  // create new 320x240 image
  ```

```
cv::Mat img(Size(320,240),CV_8UC3);
// select a roi
cv::Mat roi(img, Rect(10,10,100,100));
// fill the ROI with (0,255,0) (which is green in RGB space);
// the original 320x240 image will be modified
roi = Scalar(0,255,0);
```

Thanks to the additional `datastart` and `dataend` members, it is possible to compute the relative sub-matrix position in the main *"container"* matrix using `locateROI()`:

```
Mat A = Mat::eye(10, 10, CV_32S);
// extracts A columns, 1 (inclusive) to 3 (exclusive).
Mat B = A(Range::all(), Range(1, 3));
// extracts B rows, 5 (inclusive) to 9 (exclusive).
// that is, C ~ A(Range(5, 9), Range(1, 3))
Mat C = B(Range(5, 9), Range::all());
Size size; Point ofs;
C.locateROI(size, ofs);
// size will be (width=10,height=10) and the ofs will be (x=1, y=5)
```

As in the case of whole matrices, if you need a deep copy, use `clone()` method of the extracted sub-matrices.

- by making a header for user-allocated-data. It can be useful for

  1. processing "foreign" data using OpenCV (e.g. when you implement a DirectShow filter or a processing module for gstreamer etc.), e.g.

     ```
     void process_video_frame(const unsigned char* pixels,
                              int width, int height, int step)
     {
         cv::Mat img(height, width, CV_8UC3, pixels, step);
         cv::GaussianBlur(img, img, cv::Size(7,7), 1.5, 1.5);
     }
     ```

  2. for quick initialization of small matrices and/or super-fast element access

     ```
     double m[3][3] = {{a, b, c}, {d, e, f}, {g, h, i}};
     cv::Mat M = cv::Mat(3, 3, CV_64F, m).inv();
     ```

  partial yet very common cases of this "user-allocated data" case are conversions from CvMat and IplImage to `Mat`. For this purpose there are special constructors taking pointers to `CvMat` or `IplImage` and the optional flag indicating whether to copy the data or not.

  Backward conversion from `Mat` to `CvMat` or `IplImage` is provided via cast operators `Mat::operator CvMat() const` an `Mat::operator IplImage()`. The operators do *not* copy the data.

```
IplImage* img = cvLoadImage("greatwave.jpg", 1);
Mat mtx(img); // convert IplImage* -> cv::Mat
CvMat oldmat = mtx; // convert cv::Mat -> CvMat
CV_Assert(oldmat.cols == img->width && oldmat.rows == img->height &&
    oldmat.data.ptr == (uchar*)img->imageData && oldmat.step == img->widthStep);
```

- by using MATLAB-style matrix initializers, `zeros(), ones(), eye()`, e.g.:

```
// create a double-precision identity martix and add it to M.
M += Mat::eye(M.rows, M.cols, CV_64F);
```

- by using comma-separated initializer:

```
// create 3x3 double-precision identity matrix
Mat M = (Mat_<double>(3,3) << 1, 0, 0, 0, 1, 0, 0, 0, 1);
```

  here we first call constructor of `Mat_` class (that we describe further) with the proper matrix, and then we just put `<<` operator followed by comma-separated values that can be constants, variables, expressions etc. Also, note the extra parentheses that are needed to avoid compiler errors.

Once matrix is created, it will be automatically managed by using reference-counting mechanism (unless the matrix header is built on top of user-allocated data, in which case you should handle the data by yourself). The matrix data will be deallocated when no one points to it; if you want to release the data pointed by a matrix header before the matrix destructor is called, use `Mat::release()`.

The next important thing to learn about the matrix class is element access. Here is how the matrix is stored. The elements are stored in row-major order (row by row). The `Mat::data` member points to the first element of the first row, `Mat::rows` contains the number of matrix rows and `Mat::cols` – the number of matrix columns. There is yet another member, called `Mat::step` that is used to actually compute address of a matrix element. The `Mat::step` is needed because the matrix can be a part of another matrix or because there can some padding space in the end of each row for a proper alignment.

Given these parameters, address of the matrix element M_{ij} is computed as following:
`addr(`M_{ij}`)=M.data + M.step*i + j*M.elemSize()`
if you know the matrix element type, e.g. it is `float`, then you can use `at<>()` method:
`addr(`M_{ij}`)=&M.at<float>(i,j)`
(where & is used to convert the reference returned by `at` to a pointer). if you need to process a whole row of matrix, the most efficient way is to get the pointer to the row first, and then just use plain C operator `[]`:

```
// compute sum of positive matrix elements
// (assuming that M is double-precision matrix)
```

```
double sum=0;
for(int i = 0; i < M.rows; i++)
{
    const double* Mi = M.ptr<double>(i);
    for(int j = 0; j < M.cols; j++)
        sum += std::max(Mi[j], 0.);
}
```

Some operations, like the above one, do not actually depend on the matrix shape, they just process elements of a matrix one by one (or elements from multiple matrices that are sitting in the same place, e.g. matrix addition). Such operations are called element-wise and it makes sense to check whether all the input/output matrices are continuous, i.e. have no gaps in the end of each row, and if yes, process them as a single long row:

```
// compute sum of positive matrix elements, optimized variant
double sum=0;
int cols = M.cols, rows = M.rows;
if(M.isContinuous())
{
    cols *= rows;
    rows = 1;
}
for(int i = 0; i < rows; i++)
{
    const double* Mi = M.ptr<double>(i);
    for(int j = 0; j < cols; j++)
        sum += std::max(Mi[j], 0.);
}
```

in the case of continuous matrix the outer loop body will be executed just once, so the overhead will be smaller, which will be especially noticeable in the case of small matrices.

Finally, there are STL-style iterators that are smart enough to skip gaps between successive rows:

```
// compute sum of positive matrix elements, iterator-based variant
double sum=0;
MatConstIterator_<double> it = M.begin<double>(), it_end = M.end<double>();
for(; it != it_end; ++it)
    sum += std::max(*it, 0.);
```

The matrix iterators are random-access iterators, so they can be passed to any STL algorithm, including `std::sort()`.

Matrix Expressions

This is a list of implemented matrix operations that can be combined in arbitrary complex expressions (here *A*, *B* stand for matrices (`Mat`), *s* for a scalar (`Scalar`), α for a real-valued scalar (`double`)):

- addition, subtraction, negation: `A` $\pm$ `B`, `A` $\pm$ `s`, `s` $\pm$ `A`, $-$`A`
- scaling: `A`$*\alpha$, `A`$/\alpha$
- per-element multiplication and division: `A.mul(B)`, `A/B`, α`/A`
- matrix multiplication: `A*B`
- transposition: `A.t()` $\sim A^t$
- matrix inversion and pseudo-inversion, solving linear systems and least-squares problems: `A.inv([method])` $\sim A^{-1}$, `A.inv([method])*B` $\sim X : AX = B$
- comparison: `A` $\gtrless$ `B`, `A` $\neq$ `B`, `A` $\gtrless$ α, `A` $\neq \alpha$. The result of comparison is 8-bit single channel mask, which elements are set to 255 (if the particular element or pair of elements satisfy the condition) and 0 otherwise.
- bitwise logical operations: `A & B`, `A & s`, `A | B`, `A | s`, `A ^ B`, `A ^ s`, `~A`
- element-wise minimum and maximum: `min(A, B)`, `min(A,` α`)`, `max(A, B)`, `max(A,` α`)`
- element-wise absolute value: `abs(A)`
- cross-product, dot-product: `A.cross(B)`, `A.dot(B)`
- any function of matrix or matrices and scalars that returns a matrix or a scalar, such as cv::norm, cv::mean, cv::sum, cv::countNonZero, cv::trace, cv::determinant, cv::repeat etc.
- matrix initializers (`eye()`, `zeros()`, `ones()`), matrix comma-separated initializers, matrix constructors and operators that extract sub-matrices (see Mat description).
- `Mat_<destination_type>()` constructors to cast the result to the proper type.

Note, however, that comma-separated initializers and probably some other operations may require additional explicit `Mat()` or `Mat_<T>()` constuctor calls to resolve possible ambiguity.

Mat_

Template matrix class derived from Mat

```
template<typename _Tp> class Mat_ : public Mat
{
public:
    typedef _Tp value_type;
    typedef typename DataType<_Tp>::channel_type channel_type;
    typedef MatIterator_<_Tp> iterator;
    typedef MatConstIterator_<_Tp> const_iterator;

    Mat_();
    // equivalent to Mat(_rows, _cols, DataType<_Tp>::type)
    Mat_(int _rows, int _cols);
    // other forms of the above constructor
    Mat_(int _rows, int _cols, const _Tp& value);
    explicit Mat_(Size _size);
    Mat_(Size _size, const _Tp& value);
    // copy/conversion contructor. If m is of different type, it's converted
    Mat_(const Mat& m);
    // copy constructor
    Mat_(const Mat_& m);
    // construct a matrix on top of user-allocated data.
    // step is in bytes(!!!), regardless of the type
    Mat_(int _rows, int _cols, _Tp* _data, size_t _step=AUTO_STEP);
    // minor selection
    Mat_(const Mat_& m, const Range& rowRange, const Range& colRange);
    Mat_(const Mat_& m, const Rect& roi);
    // to support complex matrix expressions
    Mat_(const MatExpr_Base& expr);
    // makes a matrix out of Vec or std::vector. The matrix will have a single column
    template<int n> explicit Mat_(const Vec<_Tp, n>& vec);
    Mat_(const vector<_Tp>& vec, bool copyData=false);

    Mat_& operator = (const Mat& m);
    Mat_& operator = (const Mat_& m);
    // set all the elements to s.
    Mat_& operator = (const _Tp& s);

    // iterators; they are smart enough to skip gaps in the end of rows
    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;
```

```
// equivalent to Mat::create(_rows, _cols, DataType<_Tp>::type)
void create(int _rows, int _cols);
void create(Size _size);
// cross-product
Mat_ cross(const Mat_& m) const;
// to support complex matrix expressions
Mat_& operator = (const MatExpr_Base& expr);
// data type conversion
template<typename T2> operator Mat_<T2>() const;
// overridden forms of Mat::row() etc.
Mat_ row(int y) const;
Mat_ col(int x) const;
Mat_ diag(int d=0) const;
Mat_ clone() const;

// transposition, inversion, per-element multiplication
MatExpr_<...> t() const;
MatExpr_<...> inv(int method=DECOMP_LU) const;

MatExpr_<...> mul(const Mat_& m, double scale=1) const;
MatExpr_<...> mul(const MatExpr_<...>& m, double scale=1) const;

// overridden forms of Mat::elemSize() etc.
size_t elemSize() const;
size_t elemSize1() const;
int type() const;
int depth() const;
int channels() const;
size_t step1() const;
// returns step()/sizeof(_Tp)
size_t stepT() const;

// overridden forms of Mat::zeros() etc. Data type is omitted, of course
static MatExpr_Initializer zeros(int rows, int cols);
static MatExpr_Initializer zeros(Size size);
static MatExpr_Initializer ones(int rows, int cols);
static MatExpr_Initializer ones(Size size);
static MatExpr_Initializer eye(int rows, int cols);
static MatExpr_Initializer eye(Size size);

// some more overriden methods
Mat_ reshape(int _rows) const;
Mat_& adjustROI( int dtop, int dbottom, int dleft, int dright );
Mat_ operator()( const Range& rowRange, const Range& colRange ) const;
```

```
    Mat_ operator()( const Rect& roi ) const;

    // more convenient forms of row and element access operators
    _Tp* operator [](int y);
    const _Tp* operator [](int y) const;

    _Tp& operator ()(int row, int col);
    const _Tp& operator ()(int row, int col) const;
    _Tp& operator ()(Point pt);
    const _Tp& operator ()(Point pt) const;

    // to support matrix expressions
    operator MatExpr_<Mat_, Mat_>() const;

    // conversion to vector.
    operator vector<_Tp>() const;
};
```

The class `Mat_<_Tp>` is a ”thin” template wrapper on top of `Mat` class. It does not have any extra data fields, nor it or `Mat` have any virtual methods and thus references or pointers to these two classes can be freely converted one to another. But do it with care, e.g.:

```
// create 100x100 8-bit matrix
Mat M(100,100,CV_8U);
// this will compile fine. no any data conversion will be done.
Mat_<float>& M1 = (Mat_<float>&)M;
// the program will likely crash at the statement below
M1(99,99) = 1.f;
```

While `Mat` is sufficient in most cases, `Mat_` can be more convenient if you use a lot of element access operations and if you know matrix type at compile time. Note that `Mat::at<_Tp>(int y, int x)` and `Mat_<_Tp>::operator ()(int y, int x)` do absolutely the same and run at the same speed, but the latter is certainly shorter:

```
Mat_<double> M(20,20);
for(int i = 0; i < M.rows; i++)
    for(int j = 0; j < M.cols; j++)
        M(i,j) = 1./(i+j+1);
Mat E, V;
eigen(M,E,V);
cout << E.at<double>(0,0)/E.at<double>(M.rows-1,0);
```

How to use `Mat_` for multi-channel images/matrices?

This is simple - just pass `Vec` as `Mat_` parameter:

```
// allocate 320x240 color image and fill it with green (in RGB space)
Mat_<Vec3b> img(240, 320, Vec3b(0,255,0));
```

```
// now draw a diagonal white line
for(int i = 0; i < 100; i++)
    img(i,i)=Vec3b(255,255,255);
// and now scramble the 2nd (red) channel of each pixel
for(int i = 0; i < img.rows; i++)
    for(int j = 0; j < img.cols; j++)
        img(i,j)[2] ^= (uchar)(i ^ j);
```

MatND

n-dimensional dense array

```
class MatND
{
public:
    // default constructor
    MatND();
    // constructs array with specific size and data type
    MatND(int _ndims, const int* _sizes, int _type);
    // constructs array and fills it with the specified value
    MatND(int _ndims, const int* _sizes, int _type, const Scalar& _s);
    // copy constructor. only the header is copied.
    MatND(const MatND& m);
    // sub-array selection. only the header is copied
    MatND(const MatND& m, const Range* ranges);
    // converts old-style nd array to MatND; optionally, copies the data
    MatND(const CvMatND* m, bool copyData=false);
    ~MatND();
    MatND& operator = (const MatND& m);

    // creates a complete copy of the matrix (all the data is copied)
    MatND clone() const;
    // sub-array selection; only the header is copied
    MatND operator()(const Range* ranges) const;

    // copies the data to another matrix.
    // Calls m.create(this->size(), this->type()) prior to
    // copying the data
    void copyTo( MatND& m ) const;
    // copies only the selected elements to another matrix.
    void copyTo( MatND& m, const MatND& mask ) const;
    // converts data to the specified data type.
    // calls m.create(this->size(), rtype) prior to the conversion
    void convertTo( MatND& m, int rtype, double alpha=1, double beta=0 ) const;
```

```
// assigns "s" to each array element.
MatND& operator = (const Scalar& s);
// assigns "s" to the selected elements of array
// (or to all the elements if mask==MatND())
MatND& setTo(const Scalar& s, const MatND& mask=MatND());
// modifies geometry of array without copying the data
MatND reshape(int _newcn, int _newndims=0, const int* _newsz=0) const;

// allocates a new buffer for the data unless the current one already
// has the specified size and type.
void create(int _ndims, const int* _sizes, int _type);
// manually increment reference counter (use with care !!!)
void addref();
// decrements the reference counter. Deallocates the data when
// the reference counter reaches zero.
void release();

// converts the matrix to 2D Mat or to the old-style CvMatND.
// In either case the data is not copied.
operator Mat() const;
operator CvMatND() const;
// returns true if the array data is stored continuously
bool isContinuous() const;
// returns size of each element in bytes
size_t elemSize() const;
// returns size of each element channel in bytes
size_t elemSize1() const;
// returns OpenCV data type id (CV_8UC1, ... CV_64FC4,...)
int type() const;
// returns depth (CV_8U ... CV_64F)
int depth() const;
// returns the number of channels
int channels() const;
// step1() ~ step()/elemSize1()
size_t step1(int i) const;

// return pointer to the element (versions for 1D, 2D, 3D and generic nD cases)
uchar* ptr(int i0);
const uchar* ptr(int i0) const;
uchar* ptr(int i0, int i1);
const uchar* ptr(int i0, int i1) const;
uchar* ptr(int i0, int i1, int i2);
const uchar* ptr(int i0, int i1, int i2) const;
uchar* ptr(const int* idx);
```

```
    const uchar* ptr(const int* idx) const;

    // convenient template methods for element access.
    // note that _Tp must match the actual matrix type -
    // the functions do not do any on-fly type conversion
    template<typename _Tp> _Tp& at(int i0);
    template<typename _Tp> const _Tp& at(int i0) const;
    template<typename _Tp> _Tp& at(int i0, int i1);
    template<typename _Tp> const _Tp& at(int i0, int i1) const;
    template<typename _Tp> _Tp& at(int i0, int i1, int i2);
    template<typename _Tp> const _Tp& at(int i0, int i1, int i2) const;
    template<typename _Tp> _Tp& at(const int* idx);
    template<typename _Tp> const _Tp& at(const int* idx) const;

    enum { MAGIC_VAL=0x42FE0000, AUTO_STEP=-1,
        CONTINUOUS_FLAG=CV_MAT_CONT_FLAG, MAX_DIM=CV_MAX_DIM };

    // combines data type, continuity flag, signature (magic value)
    int flags;
    // the array dimensionality
    int dims;

    // data reference counter
    int* refcount;
    // pointer to the data
    uchar* data;
    // and its actual beginning and end
    uchar* datastart;
    uchar* dataend;

    // step and size for each dimension, MAX_DIM at max
    int size[MAX_DIM];
    size_t step[MAX_DIM];
};
```

The class `MatND` describes n-dimensional dense numerical single-channel or multi-channel array. This is a convenient representation for multi-dimensional histograms (when they are not very sparse, otherwise `SparseMat` will do better), voxel volumes, stacked motion fields etc. The data layout of matrix M is defined by the array of `M.step[]`, so that the address of element $(i_0, ..., i_{M.dims-1})$, where $0 \leq i_k < M.size[k]$ is computed as:

$$addr(M_{i_0,...,i_{M.dims-1}}) = M.data + M.step[0]*i_0 + M.step[1]*i_1 + ... + M.step[M.dims-1]*i_{M.dims-1}$$

which is more general form of the respective formula for Mat , wherein `size[0]` $\sim$ `rows`, `size[1]` $\sim$ `cols`, `step[0]` was simply called `step`, and `step[1]` was not stored at all but

computed as `Mat::elemSize()`.

In other aspects `MatND` is also very similar to `Mat`, with the following limitations and differences:

- much less operations are implemented for `MatND`
- currently, algebraic expressions with `MatND`'s are not supported
- the `MatND` iterator is completely different from `Mat` and `Mat_` iterators. The latter are per-element iterators, while the former is per-slice iterator, see below.

Here is how you can use `MatND` to compute NxNxN histogram of color 8bpp image (i.e. each channel value ranges from 0..255 and we quantize it to 0..N-1):

```
void computeColorHist(const Mat& image, MatND& hist, int N)
{
    const int histSize[] = {N, N, N};

    // make sure that the histogram has proper size and type
    hist.create(3, histSize, CV_32F);

    // and clear it
    hist = Scalar(0);

    // the loop below assumes that the image
    // is 8-bit 3-channel, so let's check it.
    CV_Assert(image.type() == CV_8UC3);
    MatConstIterator_<Vec3b> it = image.begin<Vec3b>(),
                             it_end = image.end<Vec3b>();
    for( ; it != it_end; ++it )
    {
        const Vec3b& pix = *it;

        // we could have incremented the cells by 1.f/(image.rows*image.cols)
        // instead of 1.f to make the histogram normalized.
        hist.at<float>(pix[0]*N/256, pix[1]*N/256, pix[2]*N/256) += 1.f;
    }
}
```

And here is how you can iterate through `MatND` elements:

```
void normalizeColorHist(MatND& hist)
{
#if 1
    // intialize iterator (the style is different from STL).
    // after initialization the iterator will contain
    // the number of slices or planes
    // the iterator will go through
```

```
    MatNDIterator it(hist);
    double s = 0;
    // iterate through the matrix. on each iteration
    // it.planes[*] (of type Mat) will be set to the current plane.
    for(int p = 0; p < it.nplanes; p++, ++it)
        s += sum(it.planes[0])[0];
    it = MatNDIterator(hist);
    s = 1./s;
    for(int p = 0; p < it.nplanes; p++, ++it)
        it.planes[0] *= s;
#elif 1
    // this is a shorter implementation of the above
    // using built-in operations on MatND
    double s = sum(hist)[0];
    hist.convertTo(hist, hist.type(), 1./s, 0);
#else
    // and this is even shorter one
    // (assuming that the histogram elements are non-negative)
    normalize(hist, hist, 1, 0, NORM_L1);
#endif
}
```

You can iterate though several matrices simultaneously as long as they have the same geometry (dimensionality and all the dimension sizes are the same), which is useful for binary and n-ary operations on such matrices. Just pass those matrices to `MatNDIterator`. Then, during the iteration `it.planes[0]`, `it.planes[1]`, ... will be the slices of the corresponding matrices.

MatND_

Template class for n-dimensional dense array derived from MatND .

```
template<typename _Tp> class MatND_ : public MatND
{
public:
    typedef _Tp value_type;
    typedef typename DataType<_Tp>::channel_type channel_type;

    // constructors, the same as in MatND, only the type is omitted
    MatND_();
    MatND_(int dims, const int* _sizes);
    MatND_(int dims, const int* _sizes, const _Tp& _s);
    MatND_(const MatND& m);
    MatND_(const MatND_& m);
    MatND_(const MatND_& m, const Range* ranges);
    MatND_(const CvMatND* m, bool copyData=false);
```

```
    MatND_& operator = (const MatND& m);
    MatND_& operator = (const MatND_& m);
    // different initialization function
    // where we take _Tp instead of Scalar
    MatND_& operator = (const _Tp& s);

    // no special destructor is needed; use the one from MatND

    void create(int dims, const int* _sizes);
    template<typename T2> operator MatND_<T2>() const;
    MatND_ clone() const;
    MatND_ operator()(const Range* ranges) const;

    size_t elemSize() const;
    size_t elemSize1() const;
    int type() const;
    int depth() const;
    int channels() const;
    // step[i]/elemSize()
    size_t stepT(int i) const;
    size_t step1(int i) const;

    // shorter alternatives for MatND::at<_Tp>.
    _Tp& operator ()(const int* idx);
    const _Tp& operator ()(const int* idx) const;
    _Tp& operator ()(int idx0);
    const _Tp& operator ()(int idx0) const;
    _Tp& operator ()(int idx0, int idx1);
    const _Tp& operator ()(int idx0, int idx1) const;
    _Tp& operator ()(int idx0, int idx1, int idx2);
    const _Tp& operator ()(int idx0, int idx1, int idx2) const;
    _Tp& operator ()(int idx0, int idx1, int idx2);
    const _Tp& operator ()(int idx0, int idx1, int idx2) const;
};
```

`MatND_` relates to `MatND` almost like `Mat_` to `Mat` - it provides a bit more convenient element access operations and adds no extra members of virtual methods to the base class, thus references/pointers to `MatND_` and `MatND` can be easily converted one to another, e.g.

```
// alternative variant of the above histogram accumulation loop
...
CV_Assert(hist.type() == CV_32FC1);
MatND_<float>& _hist = (MatND_<float>&)hist;
for( ; it != it_end; ++it )
{
    const Vec3b& pix = *it;
```

```
    _hist(pix[0]*N/256, pix[1]*N/256, pix[2]*N/256) += 1.f;
}
...
```

SparseMat

Sparse n-dimensional array.

```
class SparseMat
{
public:
    typedef SparseMatIterator iterator;
    typedef SparseMatConstIterator const_iterator;

    // internal structure - sparse matrix header
    struct Hdr
    {
        ...
    };

    // sparse matrix node - element of a hash table
    struct Node
    {
        size_t hashval;
        size_t next;
        int idx[CV_MAX_DIM];
    };

    ////////// constructors and destructor //////////
    // default constructor
    SparseMat();
    // creates matrix of the specified size and type
    SparseMat(int dims, const int* _sizes, int _type);
    // copy constructor
    SparseMat(const SparseMat& m);
    // converts dense 2d matrix to the sparse form,
    // if try1d is true and matrix is a single-column matrix (Nx1),
    // then the sparse matrix will be 1-dimensional.
    SparseMat(const Mat& m, bool try1d=false);
    // converts dense n-d matrix to the sparse form
    SparseMat(const MatND& m);
    // converts old-style sparse matrix to the new-style.
    // all the data is copied, so that "m" can be safely
    // deleted after the conversion
```

```
    SparseMat(const CvSparseMat* m);
    // destructor
    ~SparseMat();

    ///////// assignment operations ///////////

    // this is O(1) operation; no data is copied
    SparseMat& operator = (const SparseMat& m);
    // (equivalent to the corresponding constructor with try1d=false)
    SparseMat& operator = (const Mat& m);
    SparseMat& operator = (const MatND& m);

    // creates full copy of the matrix
    SparseMat clone() const;

    // copy all the data to the destination matrix.
    // the destination will be reallocated if needed.
    void copyTo( SparseMat& m ) const;
    // converts 1D or 2D sparse matrix to dense 2D matrix.
    // If the sparse matrix is 1D, then the result will
    // be a single-column matrix.
    void copyTo( Mat& m ) const;
    // converts arbitrary sparse matrix to dense matrix.
    // watch out the memory!
    void copyTo( MatND& m ) const;
    // multiplies all the matrix elements by the specified scalar
    void convertTo( SparseMat& m, int rtype, double alpha=1 ) const;
    // converts sparse matrix to dense matrix with optional type conversion and scaling.
    // When rtype=-1, the destination element type will be the same
    // as the sparse matrix element type.
    // Otherwise rtype will specify the depth and
    // the number of channels will remain the same is in the sparse matrix
    void convertTo( Mat& m, int rtype, double alpha=1, double beta=0 ) const;
    void convertTo( MatND& m, int rtype, double alpha=1, double beta=0 ) const;

    // not used now
    void assignTo( SparseMat& m, int type=-1 ) const;

    // reallocates sparse matrix. If it was already of the proper size and type,
    // it is simply cleared with clear(), otherwise,
    // the old matrix is released (using release()) and the new one is allocated.
    void create(int dims, const int* _sizes, int _type);
    // sets all the matrix elements to 0, which means clearing the hash table.
    void clear();
    // manually increases reference counter to the header.
```

```
    void addref();
    // decreses the header reference counter, when it reaches 0,
    // the header and all the underlying data are deallocated.
    void release();

    // converts sparse matrix to the old-style representation.
    // all the elements are copied.
    operator CvSparseMat*() const;
    // size of each element in bytes
    // (the matrix nodes will be bigger because of
    //  element indices and other SparseMat::Node elements).
    size_t elemSize() const;
    // elemSize()/channels()
    size_t elemSize1() const;

    // the same is in Mat and MatND
    int type() const;
    int depth() const;
    int channels() const;

    // returns the array of sizes and 0 if the matrix is not allocated
    const int* size() const;
    // returns i-th size (or 0)
    int size(int i) const;
    // returns the matrix dimensionality
    int dims() const;
    // returns the number of non-zero elements
    size_t nzcount() const;

    // compute element hash value from the element indices:
    // 1D case
    size_t hash(int i0) const;
    // 2D case
    size_t hash(int i0, int i1) const;
    // 3D case
    size_t hash(int i0, int i1, int i2) const;
    // n-D case
    size_t hash(const int* idx) const;

    // low-level element-acccess functions,
    // special variants for 1D, 2D, 3D cases and the generic one for n-D case.
    //
    // return pointer to the matrix element.
    //  if the element is there (it's non-zero), the pointer to it is returned
    //  if it's not there and createMissing=false, NULL pointer is returned
```

```
    //  if it's not there and createMissing=true, then the new element
    //    is created and initialized with 0. Pointer to it is returned
    //  If the optional hashval pointer is not NULL, the element hash value is
    //  not computed, but *hashval is taken instead.
    uchar* ptr(int i0, bool createMissing, size_t* hashval=0);
    uchar* ptr(int i0, int i1, bool createMissing, size_t* hashval=0);
    uchar* ptr(int i0, int i1, int i2, bool createMissing, size_t* hashval=0);
    uchar* ptr(const int* idx, bool createMissing, size_t* hashval=0);

    // higher-level element access functions:
    // ref<_Tp>(i0,...[,hashval]) - equivalent to *(_Tp*)ptr(i0,...true[,hashval]).
    //    always return valid reference to the element.
    //    If it's did not exist, it is created.
    // find<_Tp>(i0,...[,hashval]) - equivalent to (_const Tp*)ptr(i0,...false[,hashval]).
    //    return pointer to the element or NULL pointer if the element is not there.
    // value<_Tp>(i0,...[,hashval]) - equivalent to
    //    { const _Tp* p = find<_Tp>(i0,...[,hashval]); return p ? *p : _Tp(); }
    //    that is, 0 is returned when the element is not there.
    // note that _Tp must match the actual matrix type -
    // the functions do not do any on-fly type conversion

    // 1D case
    template<typename _Tp> _Tp& ref(int i0, size_t* hashval=0);
    template<typename _Tp> _Tp value(int i0, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(int i0, size_t* hashval=0) const;

    // 2D case
    template<typename _Tp> _Tp& ref(int i0, int i1, size_t* hashval=0);
    template<typename _Tp> _Tp value(int i0, int i1, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(int i0, int i1, size_t* hashval=0) const;

    // 3D case
    template<typename _Tp> _Tp& ref(int i0, int i1, int i2, size_t* hashval=0);
    template<typename _Tp> _Tp value(int i0, int i1, int i2, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(int i0, int i1, int i2, size_t* hashval=0) const

    // n-D case
    template<typename _Tp> _Tp& ref(const int* idx, size_t* hashval=0);
    template<typename _Tp> _Tp value(const int* idx, size_t* hashval=0) const;
    template<typename _Tp> const _Tp* find(const int* idx, size_t* hashval=0) const;

    // erase the specified matrix element.
    // When there is no such element, the methods do nothing
    void erase(int i0, int i1, size_t* hashval=0);
    void erase(int i0, int i1, int i2, size_t* hashval=0);
```

```
    void erase(const int* idx, size_t* hashval=0);

    // return the matrix iterators,
    //   pointing to the first sparse matrix element,
    SparseMatIterator begin();
    SparseMatConstIterator begin() const;
    //   ... or to the point after the last sparse matrix element
    SparseMatIterator end();
    SparseMatConstIterator end() const;

    // and the template forms of the above methods.
    // _Tp must match the actual matrix type.
    template<typename _Tp> SparseMatIterator_<_Tp> begin();
    template<typename _Tp> SparseMatConstIterator_<_Tp> begin() const;
    template<typename _Tp> SparseMatIterator_<_Tp> end();
    template<typename _Tp> SparseMatConstIterator_<_Tp> end() const;

    // return value stored in the sparse martix node
    template<typename _Tp> _Tp& value(Node* n);
    template<typename _Tp> const _Tp& value(const Node* n) const;

    ////////////// some internal-use methods ///////////////
    ...

    // pointer to the sparse matrix header
    Hdr* hdr;
};
```

The class `SparseMat` represents multi-dimensional sparse numerical arrays. Such a sparse array can store elements of any type that Mat and MatND can store. "Sparse" means that only non-zero elements are stored (though, as a result of operations on a sparse matrix, some of its stored elements can actually become 0. It's up to the user to detect such elements and delete them using `SparseMat::erase`). The non-zero elements are stored in a hash table that grows when it's filled enough, so that the search time is O(1) in average (regardless of whether element is there or not). Elements can be accessed using the following methods:

1. query operations (`SparseMat::ptr` and the higher-level `SparseMat::ref`, `SparseMat::value` and `SparseMat::find`), e.g.:

```
    const int dims = 5;
    int size[] = {10, 10, 10, 10, 10};
    SparseMat sparse_mat(dims, size, CV_32F);
    for(int i = 0; i < 1000; i++)
    {
        int idx[dims];
```

```
        for(int k = 0; k < dims; k++)
            idx[k] = rand()%sparse_mat.size(k);
        sparse_mat.ref<float>(idx) += 1.f;
    }
```

2. sparse matrix iterators. Like Mat iterators and unlike MatND iterators, the sparse matrix iterators are STL-style, that is, the iteration loop is familiar to C++ users:

```
    // prints elements of a sparse floating-point matrix
    // and the sum of elements.
    SparseMatConstIterator_<float>
        it = sparse_mat.begin<float>(),
        it_end = sparse_mat.end<float>();
    double s = 0;
    int dims = sparse_mat.dims();
    for(; it != it_end; ++it)
    {
        // print element indices and the element value
        const Node* n = it.node();
        printf("(")
        for(int i = 0; i < dims; i++)
            printf("%3d%c", n->idx[i], i < dims-1 ? ',' : ')');
        printf(": %f\n", *it);
        s += *it;
    }
    printf("Element sum is %g\n", s);
```

If you run this loop, you will notice that elements are enumerated in no any logical order (lexicographical etc.), they come in the same order as they stored in the hash table, i.e. semi-randomly. You may collect pointers to the nodes and sort them to get the proper ordering. Note, however, that pointers to the nodes may become invalid when you add more elements to the matrix; this is because of possible buffer reallocation.

3. a combination of the above 2 methods when you need to process 2 or more sparse matrices simultaneously, e.g. this is how you can compute unnormalized cross-correlation of the 2 floating-point sparse matrices:

```
    double cross_corr(const SparseMat& a, const SparseMat& b)
    {
        const SparseMat *_a = &a, *_b = &b;
        // if b contains less elements than a,
        // it's faster to iterate through b
        if(_a->nzcount() > _b->nzcount())
            std::swap(_a, _b);
        SparseMatConstIterator_<float> it = _a->begin<float>(),
```

```
                                        it_end = _a->end<float>();
        double ccorr = 0;
        for(; it != it_end; ++it)
        {
            // take the next element from the first matrix
            float avalue = *it;
            const Node* anode = it.node();
            // and try to find element with the same index in the second matrix.
            // since the hash value depends only on the element index,
            // we reuse hashvalue stored in the node
            float bvalue = _b->value<float>(anode->idx,&anode->hashval);
            ccorr += avalue*bvalue;
        }
        return ccorr;
    }
```

SparseMat_

Template sparse n-dimensional array class derived from SparseMat

```
template<typename _Tp> class SparseMat_ : public SparseMat
{
public:
    typedef SparseMatIterator_<_Tp> iterator;
    typedef SparseMatConstIterator_<_Tp> const_iterator;

    // constructors;
    // the created matrix will have data type = DataType<_Tp>::type
    SparseMat_();
    SparseMat_(int dims, const int* _sizes);
    SparseMat_(const SparseMat& m);
    SparseMat_(const SparseMat_& m);
    SparseMat_(const Mat& m);
    SparseMat_(const MatND& m);
    SparseMat_(const CvSparseMat* m);
    // assignment operators; data type conversion is done when necessary
    SparseMat_& operator = (const SparseMat& m);
    SparseMat_& operator = (const SparseMat_& m);
    SparseMat_& operator = (const Mat& m);
    SparseMat_& operator = (const MatND& m);

    // equivalent to the correspoding parent class methods
    SparseMat_ clone() const;
    void create(int dims, const int* _sizes);
```

```
    operator CvSparseMat*() const;

    // overriden methods that do extra checks for the data type
    int type() const;
    int depth() const;
    int channels() const;

    // more convenient element access operations.
    // ref() is retained (but <_Tp> specification is not need anymore);
    // operator () is equivalent to SparseMat::value<_Tp>
    _Tp& ref(int i0, size_t* hashval=0);
    _Tp operator()(int i0, size_t* hashval=0) const;
    _Tp& ref(int i0, int i1, size_t* hashval=0);
    _Tp operator()(int i0, int i1, size_t* hashval=0) const;
    _Tp& ref(int i0, int i1, int i2, size_t* hashval=0);
    _Tp operator()(int i0, int i1, int i2, size_t* hashval=0) const;
    _Tp& ref(const int* idx, size_t* hashval=0);
    _Tp operator()(const int* idx, size_t* hashval=0) const;

    // iterators
    SparseMatIterator_<_Tp> begin();
    SparseMatConstIterator_<_Tp> begin() const;
    SparseMatIterator_<_Tp> end();
    SparseMatConstIterator_<_Tp> end() const;
};
```

`SparseMat_` is a thin wrapper on top of SparseMat , made in the same way as `Mat_` and `MatND_`. It simplifies notation of some operations, and that's it.

```
int sz[] = {10, 20, 30};
SparseMat_<double> M(3, sz);
...
M.ref(1, 2, 3) = M(4, 5, 6) + M(7, 8, 9);
```

7.2 Operations on Arrays

---

cv::abs

Computes absolute value of each matrix element

```
MatExpr<...> abs(const Mat& src);
MatExpr<...> abs(const MatExpr<...>& src);
```

src matrix or matrix expression

`abs` is a meta-function that is expanded to one of cv::absdiff forms:

- `C = abs(A-B)` is equivalent to `absdiff(A, B, C)` and
- `C = abs(A)` is equivalent to `absdiff(A, Scalar::all(0), C)`.
- `C = Mat_<Vec<uchar,n> >(abs(A*α + β))` is equivalent to `convertScaleAbs(A, C, alpha, beta)`

The output matrix will have the same size and the same type as the input one (except for the last case, where `C` will be `depth=CV_8U`).

See also: Matrix Expressions , cv::absdiff, saturate_cast

---

cv::absdiff

Computes per-element absolute difference between 2 arrays or between array and a scalar.

```
void absdiff(const Mat& src1, const Mat& src2, Mat& dst);
void absdiff(const Mat& src1, const Scalar& sc, Mat& dst);
void absdiff(const MatND& src1, const MatND& src2, MatND& dst);
void absdiff(const MatND& src1, const Scalar& sc, MatND& dst);
```

src1 The first input array

src2 The second input array; Must be the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

The functions `absdiff` compute:

- absolute difference between two arrays

$$\texttt{dst}(I) = \texttt{saturate}(|\texttt{src1}(I) - \texttt{src2}(I)|)$$

- or absolute difference between array and a scalar:

$$\texttt{dst}(I) = \texttt{saturate}(|\texttt{src1}(I) - \texttt{sc}|)$$

where `I` is multi-dimensional index of array elements. in the case of multi-channel arrays each channel is processed independently.

See also: cv::abs, saturate_cast

cv::add

Computes the per-element sum of two arrays or an array and a scalar.

```
void add(const Mat& src1, const Mat& src2, Mat& dst);
void add(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask);
void add(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void add(const MatND& src1, const MatND& src2, MatND& dst);
void add(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask);
void add(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `add` compute:

- the sum of two arrays:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) + \texttt{src2}(I)) \quad \texttt{if mask}(I) \neq 0$$

- or the sum of array and a scalar:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) + \texttt{sc}) \quad \texttt{if mask}(I) \neq 0$$

where `I` is multi-dimensional index of array elements.

The first function in the above list can be replaced with matrix expressions:

```
dst = src1 + src2;
dst += src1; // equivalent to add(dst, src1, dst);
```

in the case of multi-channel arrays each channel is processed independently.

See also: cv::subtract, cv::addWeighted, cv::scaleAdd, cv::convertScale, Matrix Expressions , saturate_cast.

cv::addWeighted

Computes the weighted sum of two arrays.

```
void addWeighted(const Mat& src1, double alpha, const Mat& src2,
           double beta, double gamma, Mat& dst);
void addWeighted(const MatND& src1, double alpha, const MatND& src2,
           double beta, double gamma, MatND& dst);
```

src1 The first source array

alpha Weight for the first array elements

src2 The second source array; must have the same size and same type as `src1`

beta Weight for the second array elements

dst The destination array; it will have the same size and same type as `src1`

gamma Scalar, added to each sum

The functions `addWeighted` calculate the weighted sum of two arrays as follows:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) * \texttt{alpha} + \texttt{src2}(I) * \texttt{beta} + \texttt{gamma})$$

where `I` is multi-dimensional index of array elements.

The first function can be replaced with a matrix expression:

```
dst = src1*alpha + src2*beta + gamma;
```

In the case of multi-channel arrays each channel is processed independently.

See also: cv::add, cv::subtract, cv::scaleAdd, cv::convertScale, Matrix Expressions , saturate_cast.

cv::bitwise_and

Calculates per-element bit-wise conjunction of two arrays and an array and a scalar.

```
void bitwise_and(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat());
void bitwise_and(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void bitwise_and(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask=MatND());
void bitwise_and(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_and` compute per-element bit-wise logical conjunction:

- of two arrays
$$\texttt{dst}(I) = \texttt{src1}(I) \wedge \texttt{src2}(I) \quad \texttt{if mask}(I) \neq 0$$
- or array and a scalar:
$$\texttt{dst}(I) = \texttt{src1}(I) \wedge \texttt{sc} \quad \texttt{if mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation, and in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_not, bitwise_xor

---

cv::bitwise_not

Inverts every bit of array

```
void bitwise_not(const Mat& src, Mat& dst);
void bitwise_not(const MatND& src, MatND& dst);
```

src1 The source array

dst The destination array; it is reallocated to be of the same size and the same type as `src`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_not` compute per-element bit-wise inversion of the source array:

$$\texttt{dst}(I) = \neg \texttt{src}(I)$$

In the case of floating-point source array its machine-specific bit representation (usually IEEE754-compliant) is used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_or, bitwise_xor

---

cv::bitwise_or

Calculates per-element bit-wise disjunction of two arrays and an array and a scalar.

```
void bitwise_or(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat());
void bitwise_or(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void bitwise_or(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask=MatND());
void bitwise_or(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it is reallocated to be of the same size and the same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_or` compute per-element bit-wise logical disjunction

- of two arrays
$$\texttt{dst}(I) = \texttt{src1}(I) \vee \texttt{src2}(I) \quad \texttt{if mask}(I) \neq 0$$
- or array and a scalar:
$$\texttt{dst}(I) = \texttt{src1}(I) \vee \texttt{sc} \quad \texttt{if mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_not, bitwise_or

---

cv::bitwise_xor

Calculates per-element bit-wise "exclusive or" operation on two arrays and an array and a scalar.

```
void bitwise_xor(const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat());
void bitwise_xor(const Mat& src1, const Scalar& sc,
           Mat& dst, const Mat& mask=Mat());
void bitwise_xor(const MatND& src1, const MatND& src2,
           MatND& dst, const MatND& mask=MatND());
void bitwise_xor(const MatND& src1, const Scalar& sc,
           MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the second input parameter

dst The destination array; it is reallocated to be of the same size and the same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `bitwise_xor` compute per-element bit-wise logical "exclusive or" operation

- on two arrays
$$\texttt{dst}(I) = \texttt{src1}(I) \oplus \texttt{src2}(I) \quad \texttt{if mask}(I) \neq 0$$
- or array and a scalar:
$$\texttt{dst}(I) = \texttt{src1}(I) \oplus \texttt{sc} \quad \texttt{if mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

See also: bitwise_and, bitwise_not, bitwise_or

---

cv::calcCovarMatrix

Calculates covariation matrix of a set of vectors

```
void calcCovarMatrix( const Mat* samples, int nsamples,
           Mat& covar, Mat& mean,
           int flags, int ctype=CV_64F);
void calcCovarMatrix( const Mat& samples, Mat& covar, Mat& mean,
           int flags, int ctype=CV_64F);
```

samples The samples, stored as separate matrices, or as rows or columns of a single matrix

nsamples The number of samples when they are stored separately

covar The output covariance matrix; it will have type=`ctype` and square size

mean The input or output (depending on the flags) array - the mean (average) vector of the input vectors

flags The operation flags, a combination of the following values

CV_COVAR_SCRAMBLED The output covariance matrix is calculated as:

$$\texttt{scale} \cdot [\texttt{vects}[0] - \texttt{mean}, \texttt{vects}[1] - \texttt{mean}, ...]^T \cdot [\texttt{vects}[0] - \texttt{mean}, \texttt{vects}[1] - \texttt{mean}, ...]$$

, that is, the covariance matrix will be $\texttt{nsamples} \times \texttt{nsamples}$. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces technique for face recognition). Eigenvalues of this ”scrambled” matrix will match the eigenvalues of the true covariance matrix and the ”true” eigenvectors can be easily calculated from the eigenvectors of the ”scrambled” covariance matrix.

CV_COVAR_NORMAL The output covariance matrix is calculated as:

$$\texttt{scale} \cdot [\texttt{vects}[0] - \texttt{mean}, \texttt{vects}[1] - \texttt{mean}, ...] \cdot [\texttt{vects}[0] - \texttt{mean}, \texttt{vects}[1] - \texttt{mean}, ...]^T$$

, that is, `covar` will be a square matrix of the same size as the total number of elements in each input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

CV_COVAR_USE_AVG If the flag is specified, the function does not calculate `mean` from the input vectors, but, instead, uses the passed `mean` vector. This is useful if `mean` has been pre-computed or known a-priori, or if the covariance matrix is calculated by parts - in this case, `mean` is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.

CV_COVAR_SCALE If the flag is specified, the covariance matrix is scaled. In the "normal" mode `scale` is `1./nsamples`; in the "scrambled" mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled (i.e. `scale=1`).

CV_COVAR_ROWS [Only useful in the second variant of the function] The flag means that all the input vectors are stored as rows of the `samples` matrix. `mean` should be a single-row vector in this case.

CV_COVAR_COLS [Only useful in the second variant of the function] The flag means that all the input vectors are stored as columns of the `samples` matrix. `mean` should be a single-column vector in this case.

The functions `calcCovarMatrix` calculate the covariance matrix and, optionally, the mean vector of the set of input vectors.

See also: cv::PCA, cv::mulTransposed, cv::Mahalanobis

---

cv::cartToPolar

Calculates the magnitude and angle of 2d vectors.

```
void cartToPolar(const Mat& x, const Mat& y,
           Mat& magnitude, Mat& angle,
           bool angleInDegrees=false);
```

x The array of x-coordinates; must be single-precision or double-precision floating-point array

y The array of y-coordinates; it must have the same size and same type as `x`

magnitude The destination array of magnitudes of the same size and same type as `x`

angle The destination array of angles of the same size and same type as `x`. The angles are measured in radians (0 to 2π) or in degrees (0 to 360 degrees).

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function `cartToPolar` calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

$$\texttt{magnitude}(I) = \sqrt{\texttt{x}(I)^2 + \texttt{y}(I)^2},$$
$$\texttt{angle}(I) = \texttt{atan2}(\texttt{y}(I), \texttt{x}(I))[\cdot 180/\pi]$$

The angles are calculated with $\sim 0.3^\circ$ accuracy. For the (0,0) point, the angle is set to 0.

---

cv::checkRange

Checks every element of an input array for invalid values.

```
bool checkRange(const Mat& src, bool quiet=true, Point* pos=0,
           double minVal=-DBL_MAX, double maxVal=DBL_MAX);
bool checkRange(const MatND& src, bool quiet=true, int* pos=0,
           double minVal=-DBL_MAX, double maxVal=DBL_MAX);
```

src The array to check

quiet The flag indicating whether the functions quietly return false when the array elements are out of range, or they throw an exception.

pos The optional output parameter, where the position of the first outlier is stored. In the second function `pos`, when not NULL, must be a pointer to array of `src.dims` elements

minVal The inclusive lower boundary of valid values range

maxVal The exclusive upper boundary of valid values range

The functions `checkRange` check that every array element is neither NaN nor $\pm\infty$. When `minVal < -DBL_MAX` and `maxVal < DBL_MAX`, then the functions also check that each value is between `minVal` and `maxVal`. in the case of multi-channel arrays each channel is processed independently. If some values are out of range, position of the first outlier is stored in `pos` (when `pos` $\neq 0$), and then the functions either return false (when `quiet=true`) or throw an exception.

cv::compare

Performs per-element comparison of two arrays or an array and scalar value.

```
void compare(const Mat& src1, const Mat& src2, Mat& dst, int cmpop);
void compare(const Mat& src1, double value,
             Mat& dst, int cmpop);
void compare(const MatND& src1, const MatND& src2,
             MatND& dst, int cmpop);
void compare(const MatND& src1, double value,
             MatND& dst, int cmpop);
```

src1 The first source array

src2 The second source array; must have the same size and same type as `src1`

value The scalar value to compare each array element with

dst The destination array; will have the same size as `src1` and type=`CV_8UC1`

cmpop The flag specifying the relation between the elements to be checked

- **CMP_EQ** $\texttt{src1}(I) = \texttt{src2}(I)$ or $\texttt{src1}(I) = \texttt{value}$
- **CMP_GT** $\texttt{src1}(I) > \texttt{src2}(I)$ or $\texttt{src1}(I) > \texttt{value}$
- **CMP_GE** $\texttt{src1}(I) \geq \texttt{src2}(I)$ or $\texttt{src1}(I) \geq \texttt{value}$
- **CMP_LT** $\texttt{src1}(I) < \texttt{src2}(I)$ or $\texttt{src1}(I) < \texttt{value}$
- **CMP_LE** $\texttt{src1}(I) \leq \texttt{src2}(I)$ or $\texttt{src1}(I) \leq \texttt{value}$
- **CMP_NE** $\texttt{src1}(I) \neq \texttt{src2}(I)$ or $\texttt{src1}(I) \neq \texttt{value}$

The functions `compare` compare each element of `src1` with the corresponding element of `src2` or with real scalar `value`. When the comparison result is true, the corresponding element of destination array is set to 255, otherwise it is set to 0:

- `dst(I) = src1(I) cmpop src2(I) ? 255 : 0`
- `dst(I) = src1(I) cmpop value ? 255 : 0`

The comparison operations can be replaced with the equivalent matrix expressions:

```
Mat dst1 = src1 >= src2;
Mat dst2 = src1 < 8;
...
```

See also: cv::checkRange, cv::min, cv::max, cv::threshold, Matrix Expressions

cv::completeSymm

Copies the lower or the upper half of a square matrix to another half.

```
void completeSymm(Mat& mtx, bool lowerToUpper=false);
```

mtx Input-output floating-point square matrix

lowerToUpper If true, the lower half is copied to the upper half, otherwise the upper half is copied to the lower half

The function `completeSymm` copies the lower half of a square matrix to its another half; the matrix diagonal remains unchanged:

- $\texttt{mtx}_{ij} = \texttt{mtx}_{ji}$ for $i > j$ if `lowerToUpper=false`
- $\texttt{mtx}_{ij} = \texttt{mtx}_{ji}$ for $i < j$ if `lowerToUpper=true`

See also: cv::flip, cv::transpose

cv::convertScaleAbs

Scales, computes absolute values and converts the result to 8-bit.

```
void convertScaleAbs(const Mat& src, Mat& dst, double alpha=1, double
beta=0);
```

src The source array

dst The destination array

alpha The optional scale factor

beta The optional delta added to the scaled values

On each element of the input array the function `convertScaleAbs` performs 3 operations sequentially: scaling, taking absolute value, conversion to unsigned 8-bit type:

$$\texttt{dst}(I) = \texttt{saturate\_cast<uchar>}(|\texttt{src}(I) * \texttt{alpha} + \texttt{beta}|)$$

in the case of multi-channel arrays the function processes each channel independently. When the output is not 8-bit, the operation can be emulated by calling `Mat::convertTo` method (or by using matrix expressions) and then by computing absolute value of the result, for example:

```
Mat_<float> A(30,30);
randu(A, Scalar(-100), Scalar(100));
Mat_<float> B = A*5 + 3;
B = abs(B);
// Mat_<float> B = abs(A*5+3) will also do the job,
// but it will allocate a temporary matrix
```

See also: cv::Mat::convertTo, cv::abs

---

cv::countNonZero

Counts non-zero array elements.

```
int countNonZero( const Mat& mtx );
int countNonZero( const MatND& mtx );
```

mtx Single-channel array

The function `cvCountNonZero` returns the number of non-zero elements in mtx:

$$\sum_{I:\,\mathtt{mtx}(I)\neq 0} 1$$

See also: cv::mean, cv::meanStdDev, cv::norm, cv::minMaxLoc, cv::calcCovarMatrix

---

cv::cubeRoot

Computes cube root of the argument

```
float cubeRoot(float val);
```

val The function argument

The function `cubeRoot` computes $\sqrt[3]{\mathtt{val}}$. Negative arguments are handled correctly, *NaN* and $\pm\infty$ are not handled. The accuracy approaches the maximum possible accuracy for single-precision data.

cv::cvarrToMat

Converts CvMat, IplImage or CvMatND to cv::Mat.

```
Mat cvarrToMat(const CvArr* src, bool copyData=false, bool
allowND=true, int coiMode=0);
```

src The source `CvMat`, `IplImage` or `CvMatND`

copyData When it is false (default value), no data is copied, only the new header is created. In this case the original array should not be deallocated while the new matrix header is used. The the parameter is true, all the data is copied, then user may deallocate the original array right after the conversion

allowND When it is true (default value), then `CvMatND` is converted to `Mat` if it's possible (e.g. then the data is contiguous). If it's not possible, or when the parameter is false, the function will report an error

coiMode The parameter specifies how the IplImage COI (when set) is handled.

- If `coiMode=0`, the function will report an error if COI is set.
- If `coiMode=1`, the function will never report an error; instead it returns the header to the whole original image and user will have to check and process COI manually, see cv::extractImageCOI.

The function `cvarrToMat` converts CvMat , IplImage or CvMatND header to cv::Mat header, and optionally duplicates the underlying data. The constructed header is returned by the function.

When `copyData=false`, the conversion is done really fast (in O(1) time) and the newly created matrix header will have `refcount=0`, which means that no reference counting is done for the matrix data, and user has to preserve the data until the new header is destructed. Otherwise, when `copyData=true`, the new buffer will be allocated and managed as if you created a new matrix from scratch and copy the data there. That is, `cvarrToMat(src, true)` $\sim$ `cvarrToMat(src, false).clone()` (assuming that COI is not set). The function provides uniform way of supporting CvArr paradigm in the code that is migrated to use new-style data structures internally. The reverse transformation, from cv::Mat to CvMat or IplImage can be done by simple assignment:

```
CvMat* A = cvCreateMat(10, 10, CV_32F);
cvSetIdentity(A);
```

```
IplImage A1; cvGetImage(A, &A1);
Mat B = cvarrToMat(A);
Mat B1 = cvarrToMat(&A1);
IplImage C = B;
CvMat C1 = B1;
// now A, A1, B, B1, C and C1 are different headers
// for the same 10x10 floating-point array.
// note, that you will need to use "&"
// to pass C & C1 to OpenCV functions, e.g:
printf("%g", cvDet(&C1));
```

Normally, the function is used to convert an old-style 2D array (CvMat or IplImage) to `Mat`, however, the function can also take CvMatND on input and create cv::Mat for it, if it's possible. And for `CvMatND A` it is possible if and only if `A.dim[i].size*A.dim.step[i] == A.dim.step[i-1]` for all or for all but one `i, 0 < i < A.dims`. That is, the matrix data should be continuous or it should be representable as a sequence of continuous matrices. By using this function in this way, you can process CvMatND using arbitrary element-wise function. But for more complex operations, such as filtering functions, it will not work, and you need to convert CvMatND to cv::MatND using the corresponding constructor of the latter.

The last parameter, `coiMode`, specifies how to react on an image with COI set: by default it's 0, and then the function reports an error when an image with COI comes in. And `coiMode=1` means that no error is signaled - user has to check COI presence and handle it manually. The modern structures, such as cv::Mat and cv::MatND do not support COI natively. To process individual channel of an new-style array, you will need either to organize loop over the array (e.g. using matrix iterators) where the channel of interest will be processed, or extract the COI using cv::mixChannels (for new-style arrays) or cv::extractImageCOI (for old-style arrays), process this individual channel and insert it back to the destination array if need (using cv::mixChannel or cv::insertImageCOI, respectively).

See also: cv::cvGetImage, cv::cvGetMat, cv::cvGetMatND, cv::extractImageCOI, cv::insertImageCOI, cv::mixChannels

---

cv::dct

Performs a forward or inverse discrete cosine transform of 1D or 2D array

```
void dct(const Mat& src, Mat& dst, int flags=0);
```

src The source floating-point array

dst The destination array; will have the same size and same type as `src`

flags Transformation flags, a combination of the following values

DCT_INVERSE do an inverse 1D or 2D transform instead of the default forward transform.

DCT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

The function `dct` performs a forward or inverse discrete cosine transform (DCT) of a 1D or 2D floating-point array:

Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C_{jk}^{(N)} = \sqrt{\alpha_j / N} \cos\left(\frac{\pi(2k+1)j}{2N}\right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.

Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)}\right)^{-1} \cdot Y = \left(C^{(N)}\right)^{T} \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)}\right)^{T} = I$)

Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)}\right)^{T}$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)}\right)^{T} \cdot X \cdot C^{(N)}$$

The function chooses the mode of operation by looking at the flags and size of the input array:

- if `(flags & DCT_INVERSE) == 0`, the function does forward 1D or 2D transform, otherwise it is inverse 1D or 2D transform.
- if `(flags & DCT_ROWS)` $\neq$ `0`, the function performs 1D transform of each row.
- otherwise, if the array is a single column or a single row, the function performs 1D transform
- otherwise it performs 2D transform.

Important note: currently cv::dct supports even-size arrays (2, 4, 6 ...). For data analysis and approximation you can pad the array when necessary.

Also, the function's performance depends very much, and not monotonically, on the array size, see cv::getOptimalDFTSize. In the current implementation DCT of a vector of size `N` is computed via DFT of a vector of size `N/2`, thus the optimal DCT size $N^* \geq N$ can be computed as:

```
size_t getOptimalDCTSize(size_t N) { return 2*getOptimalDFTSize((N+1)/2); }
```

See also: cv::dft, cv::getOptimalDFTSize, cv::idct

---

cv::dft

Performs a forward or inverse Discrete Fourier transform of 1D or 2D floating-point array.

```
void dft(const Mat& src, Mat& dst, int flags=0, int nonzeroRows=0);
```

src The source array, real or complex

dst The destination array, which size and type depends on the `flags`

flags Transformation flags, a combination of the following values

- **DFT_INVERSE** do an inverse 1D or 2D transform instead of the default forward transform.
- **DFT_SCALE** scale the result: divide it by the number of array elements. Normally, it is combined with `DFT_INVERSE` .
- **DFT_ROWS** do a forward or inverse transform of every individual row of the input matrix. This flag allows the user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.
- **DFT_COMPLEX_OUTPUT** then the function performs forward transformation of 1D or 2D real array, the result, though being a complex array, has complex-conjugate symmetry (*CCS*), see the description below. Such an array can be packed into real array of the same size as input, which is the fastest option and which is what the function does by default. However, you may wish to get the full complex array (for simpler spectrum analysis etc.). Pass the flag to tell the function to produce full-size complex output array.
- **DFT_REAL_OUTPUT** then the function performs inverse transformation of 1D or 2D complex array, the result is normally a complex array of the same size. However, if the source array has conjugate-complex symmetry (for example, it is a result of forward transformation with `DFT_COMPLEX_OUTPUT` flag), then the output is real array. While the function

itself does not check whether the input is symmetrical or not, you can pass the flag and then the function will assume the symmetry and produce the real output array. Note that when the input is packed real array and inverse transformation is executed, the function treats the input as packed complex-conjugate symmetrical array, so the output will also be real array

nonzeroRows When the parameter $\neq 0$, the function assumes that only the first `nonzeroRows` rows of the input array (`DFT_INVERSE` is not set) or only the first `nonzeroRows` of the output array (`DFT_INVERSE` is set) contain non-zeros, thus the function can handle the rest of the rows more efficiently and thus save some time. This technique is very useful for computing array cross-correlation or convolution using DFT

Forward Fourier transform of 1D vector of N elements:

$$Y = F^{(N)} \cdot X,$$

where $F^{(N)}_{jk} = \exp(-2\pi ijk/N)$ and $i = \sqrt{-1}$

Inverse Fourier transform of 1D vector of N elements:

$$X' = \left(F^{(N)}\right)^{-1} \cdot Y = \left(F^{(N)}\right)^{*} \cdot y$$
$$X = (1/N) \cdot X,$$

where $F^{*} = \left(\mathrm{Re}(F^{(N)}) - \mathrm{Im}(F^{(N)})\right)^{T}$

Forward Fourier transform of 2D vector of $M \times N$ elements:

$$Y = F^{(M)} \cdot X \cdot F^{(N)}$$

Inverse Fourier transform of 2D vector of $M \times N$ elements:

$$X' = \left(F^{(M)}\right)^{*} \cdot Y \cdot \left(F^{(N)}\right)^{*}$$
$$X = \frac{1}{M \cdot N} \cdot X'$$

In the case of real (single-channel) data, the packed format called *CCS* (complex-conjugate-symmetrical) that was borrowed from IPL and used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix}
ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\
ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\
ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\
\hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots & \hdots \\
ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \hdots & \hdots & \hdots & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\
ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \hdots & \hdots & \hdots & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\
ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \hdots & \hdots & \hdots & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2}
\end{bmatrix}$$

in the case of 1D transform of real vector, the output will look as the first row of the above matrix.

So, the function chooses the operation mode depending on the flags and size of the input array:

- if DFT_ROWS is set or the input array has single row or single column then the function performs 1D forward or inverse transform (of each row of a matrix when DFT_ROWS is set, otherwise it will be 2D transform.
- if input array is real and DFT_INVERSE is not set, the function does forward 1D or 2D transform:
  - when DFT_COMPLEX_OUTPUT is set then the output will be complex matrix of the same size as input.
  - otherwise the output will be a real matrix of the same size as input. in the case of 2D transform it will use the packed format as shown above; in the case of single 1D transform it will look as the first row of the above matrix; in the case of multiple 1D transforms (when using DCT_ROWS flag) each row of the output matrix will look like the first row of the above matrix.
- otherwise, if the input array is complex and either DFT_INVERSE or DFT_REAL_OUTPUT are not set then the output will be a complex array of the same size as input and the function will perform the forward or inverse 1D or 2D transform of the whole input array or each row of the input array independently, depending on the flags DFT_INVERSE and DFT_ROWS.
- otherwise, i.e. when DFT_INVERSE is set, the input array is real, or it is complex but DFT_REAL_OUTPUT is set, the output will be a real array of the same size as input, and the function will perform 1D or 2D inverse transformation of the whole input array or each individual row, depending on the flags DFT_INVERSE and DFT_ROWS.

The scaling is done after the transformation if DFT_SCALE is set.

Unlike cv::dct, the function supports arrays of arbitrary size, but only those arrays are processed efficiently, which sizes can be factorized in a product of small prime numbers (2, 3 and 5 in the current implementation). Such an efficient DFT size can be computed using cv::getOptimalDFTSize method.

Here is the sample on how to compute DFT-based convolution of two 2D real arrays:

```
void convolveDFT(const Mat& A, const Mat& B, Mat& C)
{
    // reallocate the output array if needed
    C.create(abs(A.rows - B.rows)+1, abs(A.cols - B.cols)+1, A.type());
    Size dftSize;
    // compute the size of DFT transform
```

```
    dftSize.width = getOptimalDFTSize(A.cols + B.cols - 1);
    dftSize.height = getOptimalDFTSize(A.rows + B.rows - 1);

    // allocate temporary buffers and initialize them with 0's
    Mat tempA(dftSize, A.type(), Scalar::all(0));
    Mat tempB(dftSize, B.type(), Scalar::all(0));

    // copy A and B to the top-left corners of tempA and tempB, respectively
    Mat roiA(tempA, Rect(0,0,A.cols,A.rows));
    A.copyTo(roiA);
    Mat roiB(tempB, Rect(0,0,B.cols,B.rows));
    B.copyTo(roiB);

    // now transform the padded A & B in-place;
    // use "nonzeroRows" hint for faster processing
    dft(tempA, tempA, 0, A.rows);
    dft(tempB, tempB, 0, B.rows);

    // multiply the spectrums;
    // the function handles packed spectrum representations well
    mulSpectrums(tempA, tempB, tempA);

    // transform the product back from the frequency domain.
    // Even though all the result rows will be non-zero,
    // we need only the first C.rows of them, and thus we
    // pass nonzeroRows == C.rows
    dft(tempA, tempA, DFT_INVERSE + DFT_SCALE, C.rows);

    // now copy the result back to C.
    tempA(Rect(0, 0, C.cols, C.rows)).copyTo(C);

    // all the temporary buffers will be deallocated automatically
}
```

What can be optimized in the above sample?

- since we passed `nonzeroRows` $\neq 0$ to the forward transform calls and since we copied `A`/`B` to the top-left corners of `tempA`/`tempB`, respectively, it's not necessary to clear the whole `tempA` and `tempB`; it is only necessary to clear the `tempA.cols - A.cols` (`tempB.cols - B.cols`) rightmost columns of the matrices.
- this DFT-based convolution does not have to be applied to the whole big arrays, especially if `B` is significantly smaller than `A` or vice versa. Instead, we can compute convolution by parts. For that we need to split the destination array `C` into multiple tiles and for each tile estimate, which parts of `A` and `B` are required to compute convolution in this tile. If the tiles in `C` are too

small, the speed will decrease a lot, because of repeated work - in the ultimate case, when each tile in `C` is a single pixel, the algorithm becomes equivalent to the naive convolution algorithm. If the tiles are too big, the temporary arrays `tempA` and `tempB` become too big and there is also slowdown because of bad cache locality. So there is optimal tile size somewhere in the middle.

- if the convolution is done by parts, since different tiles in `C` can be computed in parallel, the loop can be threaded.

All of the above improvements have been implemented in cv::matchTemplate and cv::filter2D, therefore, by using them, you can get even better performance than with the above theoretically optimal implementation (though, those two functions actually compute cross-correlation, not convolution, so you will need to "flip" the kernel or the image around the center using cv::flip).

See also: cv::dct, cv::getOptimalDFTSize, cv::mulSpectrums, cv::filter2D, cv::matchTemplate, cv::flip, cv::cartToPolar, cv::magnitude, cv::phase

---

cv::divide

Performs per-element division of two arrays or a scalar by an array.

```
void divide(const Mat& src1, const Mat& src2,
            Mat& dst, double scale=1);
void divide(double scale, const Mat& src2, Mat& dst);
void divide(const MatND& src1, const MatND& src2,
            MatND& dst, double scale=1);
void divide(double scale, const MatND& src2, MatND& dst);
```

src1 The first source array

src2 The second source array; should have the same size and same type as `src1`

scale Scale factor

dst The destination array; will have the same size and same type as `src2`

The functions `divide` divide one array by another:

```
dst(I) = saturate(src1(I)*scale/src2(I))
```

or a scalar by array, when there is no `src1`:

```
dst(I) = saturate(scale/src2(I))
```

The result will have the same type as `src1`. When `src2(I)=0`, `dst(I)=0` too.
See also: cv::multiply, cv::add, cv::subtract, Matrix Expressions

---

cv::determinant

Returns determinant of a square floating-point matrix.

```
double determinant(const Mat& mtx);
```

mtx The input matrix; must have `CV_32FC1` or `CV_64FC1` type and square size

The function `determinant` computes and returns determinant of the specified matrix. For small matrices (`mtx.cols=mtx.rows<=3`) the direct method is used; for larger matrices the function uses LU factorization.

For symmetric positive-determined matrices, it is also possible to compute cv::SVD: $\texttt{mtx} = U \cdot W \cdot V^T$ and then calculate the determinant as a product of the diagonal elements of W.

See also: cv::SVD, cv::trace, cv::invert, cv::solve, Matrix Expressions

---

cv::eigen

Computes eigenvalues and eigenvectors of a symmetric matrix.

```
bool eigen(const Mat& src, Mat& eigenvalues,
           int lowindex=-1, int highindex=-1);
bool eigen(const Mat& src, Mat& eigenvalues,
           Mat& eigenvectors, int lowindex=-1,
           int highindex=-1);
```

src The input matrix; must have `CV_32FC1` or `CV_64FC1` type, square size and be symmetric: $\texttt{src}^T = \texttt{src}$

eigenvalues The output vector of eigenvalues of the same type as `src`; The eigenvalues are stored in the descending order.

eigenvectors The output matrix of eigenvectors; It will have the same size and the same type as `src`; The eigenvectors are stored as subsequent matrix rows, in the same order as the corresponding eigenvalues

lowindex Optional index of largest eigenvalue/-vector to calculate. (See below.)

highindex Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The functions `eigen` compute just eigenvalues, or eigenvalues and eigenvectors of symmetric matrix `src`:

```
src*eigenvectors(i,:)' = eigenvalues(i)*eigenvectors(i,:)' (in MATLAB notation)
```

If either low- or highindex is supplied the other is required, too. Indexing is 0-based. Example: To calculate the largest eigenvector/-value set lowindex = highindex = 0. For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

See also: cv::SVD, cv::completeSymm, cv::PCA

---

cv::exp

Calculates the exponent of every array element.

```
void exp(const Mat& src, Mat& dst);
void exp(const MatND& src, MatND& dst);
```

src The source array

dst The destination array; will have the same size and same type as `src`

The function `exp` calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}}(I)$$

The maximum relative error is about 7×10^{-6} for single-precision and less than 10^{-10} for double-precision. Currently, the function converts denormalized values to zeros on output. Special values (NaN, $\pm\infty$) are not handled.

See also: cv::log, cv::cartToPolar, cv::polarToCart, cv::phase, cv::pow, cv::sqrt, cv::magnitude

---

cv::extractImageCOI

Extract the selected image channel

```
void extractImageCOI(const CvArr* src, Mat& dst, int coi=-1);
```

src The source array. It should be a pointer to CvMat or IplImage

dst The destination array; will have single-channel, and the same size and the same depth as src

coi If the parameter is >=0, it specifies the channel to extract; If it is <0, src must be a pointer to IplImage with valid COI set - then the selected COI is extracted.

The function extractImageCOI is used to extract image COI from an old-style array and put the result to the new-style C++ matrix. As usual, the destination matrix is reallocated using Mat::create if needed.

To extract a channel from a new-style matrix, use cv::mixChannels or cv::split

See also: cv::mixChannels, cv::split, cv::merge, cv::cvarrToMat, cv::cvSetImageCOI, cv::cvGetImageCOI

---

cv::fastAtan2

Calculates the angle of a 2D vector in degrees

```
float fastAtan2(float y, float x);
```

x x-coordinate of the vector

y y-coordinate of the vector

The function fastAtan2 calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0° to 360°. The accuracy is about 0.3°.

---

cv::flip

Flips a 2D array around vertical, horizontal or both axes.

```
void flip(const Mat& src, Mat& dst, int flipCode);
```

src The source array

dst The destination array; will have the same size and same type as `src`

flipCode Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas.

The function `flip` flips the array in one of three different ways (row and column indices are 0-based):

$$\texttt{dst}_{ij} = \begin{cases} \texttt{src}_{\texttt{src.rows}-i-1,j} & \text{if } \texttt{flipCode} = 0 \\ \texttt{src}_{i,\texttt{src.cols}-j-1} & \text{if } \texttt{flipCode} > 0 \\ \texttt{src}_{\texttt{src.rows}-i-1,\texttt{src.cols}-j-1} & \text{if } \texttt{flipCode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image ($\texttt{flipCode} = 0$) to switch between top-left and bottom-left image origin, which is a typical operation in video processing in Windows.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry ($\texttt{flipCode} > 0$)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry ($\texttt{flipCode} < 0$)
- reversing the order of 1d point arrays ($\texttt{flipCode} > 0$ or $\texttt{flipCode} = 0$)

See also: cv::transpose, cv::repeat, cv::completeSymm

---

cv::gemm

Performs generalized matrix multiplication.

```
void gemm(const Mat& src1, const Mat& src2, double alpha,
          const Mat& src3, double beta, Mat& dst, int flags=0);
```

src1 The first multiplied input matrix; should have `CV_32FC1`, `CV_64FC1`, `CV_32FC2` or `CV_64FC2` type

src2 The second multiplied input matrix; should have the same type as `src1`

alpha The weight of the matrix product

src3 The third optional delta matrix added to the matrix product; should have the same type as `src1` and `src2`

beta The weight of `src3`

dst The destination matrix; It will have the proper size and the same type as input matrices

flags Operation flags:

GEMM_1_T transpose `src1`

GEMM_2_T transpose `src2`

GEMM_3_T transpose `src3`

The function performs generalized matrix multiplication and similar to the corresponding functions `*gemm` in BLAS level 3. For example, `gemm(src1, src2, alpha, src3, beta, dst, GEMM_1_T + GEMM_3_T)` corresponds to

$$\texttt{dst} = \texttt{alpha} \cdot \texttt{src1}^T \cdot \texttt{src2} + \texttt{beta} \cdot \texttt{src3}^T$$

The function can be replaced with a matrix expression, e.g. the above call can be replaced with:

```
dst = alpha*src1.t()*src2 + beta*src3.t();
```

See also: cv::mulTransposed, cv::transform, Matrix Expressions

---

cv::getConvertElem

Returns conversion function for a single pixel

```
ConvertData getConvertElem(int fromType, int toType);
ConvertScaleData getConvertScaleElem(int fromType, int toType);
typedef void (*ConvertData)(const void* from, void* to, int cn);
typedef void (*ConvertScaleData)(const void* from, void* to,
           int cn, double alpha, double beta);
```

fromType The source pixel type

toType The destination pixel type

from Callback parameter: pointer to the input pixel

to Callback parameter: pointer to the output pixel

cn Callback parameter: the number of channels; can be arbitrary, 1, 100, 100000, ...

alpha ConvertScaleData callback optional parameter: the scale factor

beta ConvertScaleData callback optional parameter: the delta or offset

The functions `getConvertElem` and `getConvertScaleElem` return pointers to the functions for converting individual pixels from one type to another. While the main function purpose is to convert single pixels (actually, for converting sparse matrices from one type to another), you can use them to convert the whole row of a dense matrix or the whole matrix at once, by setting `cn = matrix.cols*matrix.rows*matrix.channels()` if the matrix data is continuous.

See also: cv::Mat::convertTo, cv::MatND::convertTo, cv::SparseMat::convertTo

---

cv::getOptimalDFTSize

Returns optimal DFT size for a given vector size.

```
int getOptimalDFTSize(int vecsize);
```

vecsize Vector size

DFT performance is not a monotonic function of a vector size, therefore, when you compute convolution of two arrays or do a spectral analysis of array, it usually makes sense to pad the input data with zeros to get a bit larger array that can be transformed much faster than the original one. Arrays, which size is a power-of-two (2, 4, 8, 16, 32, ...) are the fastest to process, though, the arrays, which size is a product of 2's, 3's and 5's (e.g. 300 = 5*5*3*2*2), are also processed quite efficiently.

The function `getOptimalDFTSize` returns the minimum number `N` that is greater than or equal to `vecsize`, such that the DFT of a vector of size `N` can be computed efficiently. In the current implementation $N = 2^p \times 3^q \times 5^r$, for some p, q, r.

The function returns a negative number if `vecsize` is too large (very close to `INT_MAX`).

While the function cannot be used directly to estimate the optimal vector size for DCT transform (since the current DCT implementation supports only even-size vectors), it can be easily computed as `getOptimalDFTSize((vecsize+1)/2)*2`.

See also: cv::dft, cv::dct, cv::idft, cv::idct, cv::mulSpectrums

cv::idct

Computes inverse Discrete Cosine Transform of a 1D or 2D array

```
void idct(const Mat& src, Mat& dst, int flags=0);
```

src The source floating-point single-channel array

dst The destination array. Will have the same size and same type as `src`

flags The operation flags.

`idct(src, dst, flags)` is equivalent to `dct(src, dst, flags | DCT_INVERSE)`. See cv::dct for details.

See also: cv::dct, cv::dft, cv::idft, cv::getOptimalDFTSize

cv::idft

Computes inverse Discrete Fourier Transform of a 1D or 2D array

```
void idft(const Mat& src, Mat& dst, int flags=0, int outputRows=0);
```

src The source floating-point real or complex array

dst The destination array, which size and type depends on the `flags`

flags The operation flags. See cv::dft

nonzeroRows The number of `dst` rows to compute. The rest of the rows will have undefined content. See the convolution sample in cv::dft description

`idft(src, dst, flags)` is equivalent to `dct(src, dst, flags | DFT_INVERSE)`. See cv::dft for details. Note, that none of `dft` and `idft` scale the result by default. Thus, you should pass `DFT_SCALE` to one of `dft` or `idft` explicitly to make these transforms mutually inverse.

See also: cv::dft, cv::dct, cv::idct, cv::mulSpectrums, cv::getOptimalDFTSize

cv::inRange

Checks if array elements lie between the elements of two other arrays.

```
void inRange(const Mat& src, const Mat& lowerb,
           const Mat& upperb, Mat& dst);
void inRange(const Mat& src, const Scalar& lowerb,
           const Scalar& upperb, Mat& dst);
void inRange(const MatND& src, const MatND& lowerb,
           const MatND& upperb, MatND& dst);
void inRange(const MatND& src, const Scalar& lowerb,
           const Scalar& upperb, MatND& dst);
```

src The first source array

lowerb The inclusive lower boundary array of the same size and type as `src`

upperb The exclusive upper boundary array of the same size and type as `src`

dst The destination array, will have the same size as `src` and `CV_8U` type

The functions `inRange` do the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lowerb}(I)_0 \leq \texttt{src}(I)_0 < \texttt{upperb}(I)_0$$

for single-channel arrays,

$$\texttt{dst}(I) = \texttt{lowerb}(I)_0 \leq \texttt{src}(I)_0 < \texttt{upperb}(I)_0 \wedge \texttt{lowerb}(I)_1 \leq \texttt{src}(I)_1 < \texttt{upperb}(I)_1$$

for two-channel arrays and so forth. `dst`(I) is set to 255 (all `1`-bits) if `src`(I) is within the specified range and 0 otherwise.

cv::invert

Finds the inverse or pseudo-inverse of a matrix

```
double invert(const Mat& src, Mat& dst, int method=DECOMP_LU);
```

src The source floating-point $M \times N$ matrix

dst The destination matrix; will have $N \times M$ size and the same type as `src`

flags The inversion method :

DECOMP_LU Gaussian elimination with optimal pivot element chosen

DECOMP_SVD Singular value decomposition (SVD) method

DECOMP_CHOLESKY Cholesky decomposion. The matrix must be symmetrical and positively defined

The function `invert` inverts matrix `src` and stores the result in `dst`. When the matrix `src` is singular or non-square, the function computes the pseudo-inverse matrix, i.e. the matrix `dst`, such that $\|\texttt{src} \cdot \texttt{dst} - I\|$ is minimal.

In the case of `DECOMP_LU` method, the function returns the `src` determinant (`src` must be square). If it is 0, the matrix is not inverted and `dst` is filled with zeros.

In the case of `DECOMP_SVD` method, the function returns the inversed condition number of `src` (the ratio of the smallest singular value to the largest singular value) and 0 if `src` is singular. The SVD method calculates a pseudo-inverse matrix if `src` is singular.

Similarly to `DECOMP_LU`, the method `DECOMP_CHOLESKY` works only with non-singular square matrices. In this case the function stores the inverted matrix in `dst` and returns non-zero, otherwise it returns 0.

See also: cv::solve, cv::SVD

---

cv::log

Calculates the natural logarithm of every array element.

```
void log(const Mat& src, Mat& dst);
void log(const MatND& src, MatND& dst);
```

src The source array

dst The destination array; will have the same size and same type as `src`

The function `log` calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}(I) = \begin{cases} \log|\texttt{src}(I)| & \text{if } \texttt{src}(I) \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where C is a large negative number (about -700 in the current implementation). The maximum relative error is about 7×10^{-6} for single-precision input and less than 10^{-10} for double-precision input. Special values (NaN, $\pm\infty$) are not handled.

See also: cv::exp, cv::cartToPolar, cv::polarToCart, cv::phase, cv::pow, cv::sqrt, cv::magnitude

---

cv::LUT

Performs a look-up table transform of an array.

```
void LUT(const Mat& src, const Mat& lut, Mat& dst);
```

src Source array of 8-bit elements

lut Look-up table of 256 elements. In the case of multi-channel source array, the table should either have a single channel (in this case the same table is used for all channels) or the same number of channels as in the source array

dst Destination array; will have the same size and the same number of channels as src, and the same depth as lut

The function LUT fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of src as follows:

$$\texttt{dst}(I) \leftarrow \texttt{lut}(\texttt{src}(I) + d)$$

where

$$d = \begin{cases} 0 & \text{if src has depth CV\_8U} \\ 128 & \text{if src has depth CV\_8S} \end{cases}$$

See also: cv::convertScaleAbs, Mat::convertTo

---

cv::magnitude

Calculates magnitude of 2D vectors.

```
void magnitude(const Mat& x, const Mat& y, Mat& magnitude);
```

x The floating-point array of x-coordinates of the vectors

y The floating-point array of y-coordinates of the vectors; must have the same size as x

dst The destination array; will have the same size and same type as x

The function magnitude calculates magnitude of 2D vectors formed from the corresponding elements of x and y arrays:

$$\texttt{dst}(I) = \sqrt{\texttt{x}(I)^2 + \texttt{y}(I)^2}$$

See also: cv::cartToPolar, cv::polarToCart, cv::phase, cv::sqrt

---

cv::Mahalanobis

Calculates the Mahalanobis distance between two vectors.

```
double Mahalanobis(const Mat& vec1, const Mat& vec2,
           const Mat& icovar);
```

vec1 The first 1D source vector

vec2 The second 1D source vector

icovar The inverse covariance matrix

The function cvMahalonobis calculates and returns the weighted distance between two vectors:

$$d(\texttt{vec1}, \texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the cv::calcCovarMatrix function and then inverted using the cv::invert function (preferably using DECOMP_SVD method, as the most accurate).

---

cv::max

Calculates per-element maximum of two arrays or array and a scalar

```
Mat_Expr<...> max(const Mat& src1, const Mat& src2);
Mat_Expr<...> max(const Mat& src1, double value);
Mat_Expr<...> max(double value, const Mat& src1);
void max(const Mat& src1, const Mat& src2, Mat& dst);
void max(const Mat& src1, double value, Mat& dst);
void max(const MatND& src1, const MatND& src2, MatND& dst);
void max(const MatND& src1, double value, MatND& dst);
```

src1 The first source array

src2 The second source array of the same size and type as `src1`

value The real scalar value

dst The destination array; will have the same size and type as `src1`

The functions `max` compute per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

or array and a scalar:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{value})$$

In the second variant, when the source array is multi-channel, each channel is compared with `value` independently.

The first 3 variants of the function listed above are actually a part of Matrix Expressions , they return the expression object that can be further transformed, or assigned to a matrix, or passed to a function etc.

See also: cv::min, cv::compare, cv::inRange, cv::minMaxLoc, Matrix Expressions

---

cv::mean

Calculates average (mean) of array elements

```
Scalar mean(const Mat& mtx);
Scalar mean(const Mat& mtx, const Mat& mask);
Scalar mean(const MatND& mtx);
Scalar mean(const MatND& mtx, const MatND& mask);
```

mtx The source array; it should have 1 to 4 channels (so that the result can be stored in cv::Scalar)

mask The optional operation mask

The functions `mean` compute mean value `M` of array elements, independently for each channel, and return it:

$$N = \sum_{I:\, \texttt{mask}(I) \neq 0} 1$$
$$M_c = \left(\sum_{I:\, \texttt{mask}(I) \neq 0} \texttt{mtx}(I)_c \right) / N$$

When all the mask elements are 0's, the functions return `Scalar::all(0)`.
See also: cv::countNonZero, cv::meanStdDev, cv::norm, cv::minMaxLoc

---

cv::meanStdDev

Calculates mean and standard deviation of array elements

```
void meanStdDev(const Mat& mtx, Scalar& mean,
           Scalar& stddev, const Mat& mask=Mat());
void meanStdDev(const MatND& mtx, Scalar& mean,
           Scalar& stddev, const MatND& mask=MatND());
```

mtx The source array; it should have 1 to 4 channels (so that the results can be stored in cv::Scalar's)

mean The output parameter: computed mean value

stddev The output parameter: computed standard deviation

mask The optional operation mask

The functions `meanStdDev` compute the mean and the standard deviation `M` of array elements, independently for each channel, and return it via the output parameters:

$$N = \sum_{I, \texttt{mask}(I) \neq 0} 1$$
$$\texttt{mean}_c = \frac{\sum_{I:\, \texttt{mask}(I) \neq 0} \texttt{src}(I)_c}{N}$$
$$\texttt{stddev}_c = \sqrt{\sum_{I:\, \texttt{mask}(I) \neq 0} \left(\texttt{src}(I)_c - \texttt{mean}_c \right)^2}$$

When all the mask elements are 0's, the functions return `mean=stddev=Scalar::all(0)`. Note that the computed standard deviation is only the diagonal of the complete normalized covariance matrix. If the full matrix is needed, you can reshape the multi-channel array $M \times N$ to the

single-channel array $M*N \times \texttt{mtx.channels()}$ (only possible when the matrix is continuous) and then pass the matrix to cv::calcCovarMatrix.

See also: cv::countNonZero, cv::mean, cv::norm, cv::minMaxLoc, cv::calcCovarMatrix

---

cv::merge

Composes a multi-channel array from several single-channel arrays.

```
void merge(const Mat* mv, size_t count, Mat& dst);
void merge(const vector<Mat>& mv, Mat& dst);
void merge(const MatND* mv, size_t count, MatND& dst);
void merge(const vector<MatND>& mv, MatND& dst);
```

mv The source array or vector of the single-channel matrices to be merged. All the matrices in `mv` must have the same size and the same type

count The number of source matrices when `mv` is a plain C array; must be greater than zero

dst The destination array; will have the same size and the same depth as `mv[0]`, the number of channels will match the number of source matrices

The functions `merge` merge several single-channel arrays (or rather interleave their elements) to make a single multi-channel array.

$$\texttt{dst}(I)_c = \texttt{mv}[c](I)$$

The function cv::split does the reverse operation and if you need to merge several multi-channel images or shuffle channels in some other advanced way, use cv::mixChannels

See also: cv::mixChannels, cv::split, cv::reshape

---

cv::min

Calculates per-element minimum of two arrays or array and a scalar

```
Mat_Expr<...> min(const Mat& src1, const Mat& src2);
Mat_Expr<...> min(const Mat& src1, double value);
Mat_Expr<...> min(double value, const Mat& src1);
void min(const Mat& src1, const Mat& src2, Mat& dst);
```

```
void min(const Mat& src1, double value, Mat& dst);
void min(const MatND& src1, const MatND& src2, MatND& dst);
void min(const MatND& src1, double value, MatND& dst);
```

src1 The first source array

src2 The second source array of the same size and type as `src1`

value The real scalar value

dst The destination array; will have the same size and type as `src1`

The functions `min` compute per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

or array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{value})$$

In the second variant, when the source array is multi-channel, each channel is compared with `value` independently.

The first 3 variants of the function listed above are actually a part of Matrix Expressions , they return the expression object that can be further transformed, or assigned to a matrix, or passed to a function etc.

See also: cv::max, cv::compare, cv::inRange, cv::minMaxLoc, Matrix Expressions

---

cv::minMaxLoc

Finds global minimum and maximum in a whole array or sub-array

```
void minMaxLoc(const Mat& src, double* minVal,
           double* maxVal=0, Point* minLoc=0,
           Point* maxLoc=0, const Mat& mask=Mat());
void minMaxLoc(const MatND& src, double* minVal,
           double* maxVal, int* minIdx=0, int* maxIdx=0,
           const MatND& mask=MatND());
void minMaxLoc(const SparseMat& src, double* minVal,
           double* maxVal, int* minIdx=0, int* maxIdx=0);
```

src The source single-channel array

minVal Pointer to returned minimum value; `NULL` if not required

maxVal Pointer to returned maximum value; `NULL` if not required

minLoc Pointer to returned minimum location (in 2D case); `NULL` if not required

maxLoc Pointer to returned maximum location (in 2D case); `NULL` if not required

minIdx Pointer to returned minimum location (in nD case); `NULL` if not required, otherwise must point to an array of `src.dims` elements and the coordinates of minimum element in each dimensions will be stored sequentially there.

maxIdx Pointer to returned maximum location (in nD case); `NULL` if not required

mask The optional mask used to select a sub-array

The functions `ninMaxLoc` find minimum and maximum element values and their positions. The extremums are searched across the whole array, or, if `mask` is not an empty array, in the specified array region.

The functions do not work with multi-channel arrays. If you need to find minimum or maximum elements across all the channels, use cv::reshape first to reinterpret the array as single-channel. Or you may extract the particular channel using cv::extractImageCOI or cv::mixChannels or cv::split.

in the case of a sparse matrix the minimum is found among non-zero elements only.

See also: cv::max, cv::min, cv::compare, cv::inRange, cv::extractImageCOI, cv::mixChannels, cv::split, cv::reshape.

---

cv::mixChannels

Copies specified channels from input arrays to the specified channels of output arrays

```
void mixChannels(const Mat* srcv, int nsrc, Mat* dstv, int ndst,
           const int* fromTo, size_t npairs);
void mixChannels(const MatND* srcv, int nsrc, MatND* dstv, int ndst,
           const int* fromTo, size_t npairs);
void mixChannels(const vector<Mat>& srcv, vector<Mat>& dstv,
           const int* fromTo, int npairs);
void mixChannels(const vector<MatND>& srcv, vector<MatND>& dstv,
           const int* fromTo, int npairs);
```

srcv The input array or vector of matrices. All the matrices must have the same size and the same depth

nsrc The number of elements in `srcv`

dstv The output array or vector of matrices. All the matrices *must be allocated*, their size and depth must be the same as in `srcv[0]`

ndst The number of elements in `dstv`

fromTo The array of index pairs, specifying which channels are copied and where. `fromTo[k*2]` is the 0-based index of the input channel in `srcv` and `fromTo[k*2+1]` is the index of the output channel in `dstv`. Here the continuous channel numbering is used, that is, the first input image channels are indexed from `0` to `srcv[0].channels()-1`, the second input image channels are indexed from `srcv[0].channels()` to `srcv[0].channels() + srcv[1].channels()-1` etc., and the same scheme is used for the output image channels. As a special case, when `fromTo[k*2]` is negative, the corresponding output channel is filled with zero. `npairs`The number of pairs. In the latter case the parameter is not passed explicitly, but computed as `srcv.size()` (=`dstv.size()`)

The functions `mixChannels` provide an advanced mechanism for shuffling image channels. cv::split and cv::merge and some forms of cv::cvtColor are partial cases of `mixChannels`.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B channels swapped) and separate alpha channel image:

```
Mat rgba( 100, 100, CV_8UC4, Scalar(1,2,3,4) );
Mat bgr( rgba.rows, rgba.cols, CV_8UC3 );
Mat alpha( rgba.rows, rgba.cols, CV_8UC1 );

// forming array of matrices is quite efficient operations,
// because the matrix data is not copied, only the headers
Mat out[] = { bgr, alpha };
// rgba[0] -> bgr[2], rgba[1] -> bgr[1],
// rgba[2] -> bgr[0], rgba[3] -> alpha[0]
int from_to[] = { 0,2,  1,1,  2,0,  3,3 };
mixChannels( &rgba, 1, out, 2, from_to, 4 );
```

Note that, unlike many other new-style C++ functions in OpenCV (see the introduction section and cv::Mat::create), `mixChannels` requires the destination arrays be pre-allocated before calling the function.

See also: cv::split, cv::merge, cv::cvtColor

cv::mulSpectrums

Performs per-element multiplication of two Fourier spectrums.

```
void mulSpectrums(const Mat& src1, const Mat& src2, Mat& dst,
           int flags, bool conj=false);
```

src1 The first source array

src2 The second source array; must have the same size and the same type as `src1`

dst The destination array; will have the same size and the same type as `src1`

flags The same flags as passed to cv::dft; only the flag `DFT_ROWS` is checked for

conj The optional flag that conjugate the second source array before the multiplication (true) or not (false)

The function `mulSpectrums` performs per-element multiplication of the two CCS-packed or complex matrices that are results of a real or complex Fourier transform.

The function, together with cv::dft and cv::idft, may be used to calculate convolution (pass `conj=false`) or correlation (pass `conj=false`) of two arrays rapidly. When the arrays are complex, they are simply multiplied (per-element) with optional conjugation of the second array elements. When the arrays are real, they assumed to be CCS-packed (see cv::dft for details).

cv::multiply

Calculates the per-element scaled product of two arrays

```
void multiply(const Mat& src1, const Mat& src2,
           Mat& dst, double scale=1);
void multiply(const MatND& src1, const MatND& src2,
           MatND& dst, double scale=1);
```

src1 The first source array

src2 The second source array of the same size and the same type as `src1`

dst The destination array; will have the same size and the same type as `src1`

scale The optional scale factor

The function `multiply` calculates the per-element product of two arrays:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I))$$

There is also Matrix Expressions -friendly variant of the first function, see cv::Mat::mul.
If you are looking for a matrix product, not per-element product, see cv::gemm.
See also: cv::add, cv::substract, cv::divide, Matrix Expressions , cv::scaleAdd, cv::addWeighted, cv::accumulate, cv::accumulateProduct, cv::accumulateSquare, cv::Mat::convertTo

---

cv::mulTransposed

Calculates the product of a matrix and its transposition.

```
void mulTransposed( const Mat& src, Mat& dst, bool aTa,
           const Mat& delta=Mat(),
           double scale=1, int rtype=-1 );
```

src The source matrix

dst The destination square matrix

aTa Specifies the multiplication ordering; see the description below

delta The optional delta matrix, subtracted from `src` before the multiplication. When the matrix is empty (`delta=Mat()`), it's assumed to be zero, i.e. nothing is subtracted, otherwise if it has the same size as `src`, then it's simply subtracted, otherwise it is "repeated" (see cv::repeat) to cover the full `src` and then subtracted. Type of the delta matrix, when it's not empty, must be the same as the type of created destination matrix, see the `rtype` description

scale The optional scale factor for the matrix product

rtype When it's negative, the destination matrix will have the same type as `src`. Otherwise, it will have `type=CV_MAT_DEPTH(rtype)`, which should be either `CV_32F` or `CV_64F`

The function `mulTransposed` calculates the product of `src` and its transposition:

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})^T(\texttt{src} - \texttt{delta})$$

if `aTa=true`, and

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})(\texttt{src} - \texttt{delta})^T$$

otherwise. The function is used to compute covariance matrix and with zero delta can be used as a faster substitute for general matrix product $A * B$ when $B = A^T$.

See also: cv::calcCovarMatrix, cv::gemm, cv::repeat, cv::reduce

---

cv::norm

Calculates absolute array norm, absolute difference norm, or relative difference norm.

```
double norm(const Mat& src1, int normType=NORM_L2);
double norm(const Mat& src1, const Mat& src2, int normType=NORM_L2);
double norm(const Mat& src1, int normType, const Mat& mask);
double norm(const Mat& src1, const Mat& src2,
           int normType, const Mat& mask);
double norm(const MatND& src1, int normType=NORM_L2,
           const MatND& mask=MatND());
double norm(const MatND& src1, const MatND& src2,
           int normType=NORM_L2, const MatND& mask=MatND());
double norm( const SparseMat& src, int normType );
```

src1 The first source array

src2 The second source array of the same size and the same type as `src1`

normType Type of the norm; see the discussion below

mask The optional operation mask

The functions `norm` calculate the absolute norm of `src1` (when there is no `src2`):

$$norm = \begin{cases} \|\texttt{src1}\|_{L_\infty} = \max_I |\texttt{src1}(I)| & \text{if } \texttt{normType} = \texttt{NORM\_INF} \\ \|\texttt{src1}\|_{L_1} = \sum_I |\texttt{src1}(I)| & \text{if } \texttt{normType} = \texttt{NORM\_L1} \\ \|\texttt{src1}\|_{L_2} = \sqrt{\sum_I \texttt{src1}(I)^2} & \text{if } \texttt{normType} = \texttt{NORM\_L2} \end{cases}$$

or an absolute or relative difference norm if src2 is there:

$$norm = \begin{cases} \|\texttt{src1} - \texttt{src2}\|_{L_\infty} = \max_I |\texttt{src1}(I) - \texttt{src2}(I)| & \text{if } \texttt{normType} = \texttt{NORM\_INF} \\ \|\texttt{src1} - \texttt{src2}\|_{L_1} = \sum_I |\texttt{src1}(I) - \texttt{src2}(I)| & \text{if } \texttt{normType} = \texttt{NORM\_L1} \\ \|\texttt{src1} - \texttt{src2}\|_{L_2} = \sqrt{\sum_I (\texttt{src1}(I) - \texttt{src2}(I))^2} & \text{if } \texttt{normType} = \texttt{NORM\_L2} \end{cases}$$

or

$$norm = \begin{cases} \frac{\|\texttt{src1} - \texttt{src2}\|_{L_\infty}}{\|\texttt{src2}\|_{L_\infty}} & \text{if } \texttt{normType} = \texttt{NORM\_RELATIVE\_INF} \\ \frac{\|\texttt{src1} - \texttt{src2}\|_{L_1}}{\|\texttt{src2}\|_{L_1}} & \text{if } \texttt{normType} = \texttt{NORM\_RELATIVE\_L1} \\ \frac{\|\texttt{src1} - \texttt{src2}\|_{L_2}}{\|\texttt{src2}\|_{L_2}} & \text{if } \texttt{normType} = \texttt{NORM\_RELATIVE\_L2} \end{cases}$$

The functions norm return the calculated norm.

When there is mask parameter, and it is not empty (then it should have type CV_8U and the same size as src1), the norm is computed only over the specified by the mask region.

A multiple-channel source arrays are treated as a single-channel, that is, the results for all channels are combined.

---

cv::normalize

Normalizes array's norm or the range

```
void normalize( const Mat& src, Mat& dst,
           double alpha=1, double beta=0,
           int normType=NORM_L2, int rtype=-1,
           const Mat& mask=Mat());
void normalize( const MatND& src, MatND& dst,
           double alpha=1, double beta=0,
           int normType=NORM_L2, int rtype=-1,
           const MatND& mask=MatND());
void normalize( const SparseMat& src, SparseMat& dst,
           double alpha, int normType );
```

src The source array

dst The destination array; will have the same size as src

alpha The norm value to normalize to or the lower range boundary in the case of range normalization

beta The upper range boundary in the case of range normalization; not used for norm normalization

normType The normalization type, see the discussion

rtype When the parameter is negative, the destination array will have the same type as `src`, otherwise it will have the same number of channels as `src` and the depth=`CV_MAT_DEPTH(rtype)`

mask The optional operation mask

The functions `normalize` scale and shift the source array elements, so that

$$\|\texttt{dst}\|_{L_p} = \texttt{alpha}$$

(where $p = \infty$, 1 or 2) when `normType=NORM_INF`, `NORM_L1` or `NORM_L2`, or so that

$$\min_I \texttt{dst}(I) = \texttt{alpha}, \ \max_I \texttt{dst}(I) = \texttt{beta}$$

when `normType=NORM_MINMAX` (for dense arrays only).

The optional mask specifies the sub-array to be normalize, that is, the norm or min-n-max are computed over the sub-array and then this sub-array is modified to be normalized. If you want to only use the mask to compute the norm or min-max, but modify the whole array, you can use cv::norm and cv::Mat::convertScale/ cv::MatND::convertScale/crossSparseMat::convertScale separately.

in the case of sparse matrices, only the non-zero values are analyzed and transformed. Because of this, the range transformation for sparse matrices is not allowed, since it can shift the zero level.

See also: cv::norm, cv::Mat::convertScale, cv::MatND::convertScale, cv::SparseMat::convertScale

---

PCA

Class for Principal Component Analysis

```
class PCA
{
public:
    // default constructor
    PCA();newline
    // computes PCA for a set of vectors stored as data rows or columns.
    PCA(const Mat& data, const Mat& mean, int flags, int maxComponents=0);newline
    // computes PCA for a set of vectors stored as data rows or columns
    PCA& operator()(const Mat& data, const Mat& mean, int flags, int maxComponents=0);newli
    // projects vector into the principal components space
```

```
    Mat project(const Mat& vec) const;newline
    void project(const Mat& vec, Mat& result) const;newline
    // reconstructs the vector from its PC projection
    Mat backProject(const Mat& vec) const;newline
    void backProject(const Mat& vec, Mat& result) const;newline

    // eigenvectors of the PC space, stored as the matrix rows
    Mat eigenvectors;newline
    // the corresponding eigenvalues; not used for PCA compression/decompression
    Mat eigenvalues;newline
    // mean vector, subtracted from the projected vector
    // or added to the reconstructed vector
    Mat mean;
};
```

The class `PCA` is used to compute the special basis for a set of vectors. The basis will consist of eigenvectors of the covariance matrix computed from the input set of vectors. And also the class `PCA` can transform vectors to/from the new coordinate space, defined by the basis. Usually, in this new coordinate system each vector from the original set (and any linear combination of such vectors) can be quite accurately approximated by taking just the first few its components, corresponding to the eigenvectors of the largest eigenvalues of the covariance matrix. Geometrically it means that we compute projection of the vector to a subspace formed by a few eigenvectors corresponding to the dominant eigenvalues of the covariation matrix. And usually such a projection is very close to the original vector. That is, we can represent the original vector from a high-dimensional space with a much shorter vector consisting of the projected vector's coordinates in the subspace. Such a transformation is also known as Karhunen-Loeve Transform, or KLT. See http://en.wikipedia.org/wiki/Principal_component_analysis

The following sample is the function that takes two matrices. The first one stores the set of vectors (a row per vector) that is used to compute PCA, the second one stores another "test" set of vectors (a row per vector) that are first compressed with PCA, then reconstructed back and then the reconstruction error norm is computed and printed for each vector.

```
PCA compressPCA(const Mat& pcaset, int maxComponents,
                const Mat& testset, Mat& compressed)
{
    PCA pca(pcaset, // pass the data
            Mat(), // we do not have a pre-computed mean vector,
                   // so let the PCA engine to compute it
            CV_PCA_DATA_AS_ROW, // indicate that the vectors
                                // are stored as matrix rows
                                // (use CV_PCA_DATA_AS_COL if the vectors are
                                // the matrix columns)
            maxComponents // specify, how many principal components to retain
            );
```

```
    // if there is no test data, just return the computed basis, ready-to-use
    if( !testset.data )
        return pca;
    CV_Assert( testset.cols == pcaset.cols );

    compressed.create(testset.rows, maxComponents, testset.type());

    Mat reconstructed;
    for( int i = 0; i < testset.rows; i++ )
    {
        Mat vec = testset.row(i), coeffs = compressed.row(i);
        // compress the vector, the result will be stored
        // in the i-th row of the output matrix
        pca.project(vec, coeffs);
        // and then reconstruct it
        pca.backProject(coeffs, reconstructed);
        // and measure the error
        printf("%d. diff = %g\n", i, norm(vec, reconstructed, NORM_L2));
    }
    return pca;
}
```

See also: cv::calcCovarMatrix, cv::mulTransposed, cv::SVD, cv::dft, cv::dct

---

cv::perspectiveTransform

Performs perspective matrix transformation of vectors.

```
void perspectiveTransform(const Mat& src,
           Mat& dst, const Mat& mtx );
```

src The source two-channel or three-channel floating-point array; each element is 2D/3D vector to be transformed

dst The destination array; it will have the same size and same type as `src`

mtx 3×3 or 4×4 transformation matrix

The function `perspectiveTransform` transforms every element of `src`, by treating it as 2D or 3D vector, in the following way (here 3D vector transformation is shown; in the case of 2D vector transformation the z component is omitted):

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Note that the function transforms a sparse set of 2D or 3D vectors. If you want to transform an image using perspective transformation, use cv::warpPerspective. If you have an inverse task, i.e. want to compute the most probable perspective transformation out of several pairs of corresponding points, you can use cv::getPerspectiveTransform or cv::findHomography.

See also: cv::transform, cv::warpPerspective, cv::getPerspectiveTransform, cv::findHomography

---

cv::phase

Calculates the rotation angle of 2d vectors

```
void phase(const Mat& x, const Mat& y, Mat& angle,
           bool angleInDegrees=false);
```

x The source floating-point array of x-coordinates of 2D vectors

y The source array of y-coordinates of 2D vectors; must have the same size and the same type as x

angle The destination array of vector angles; it will have the same size and same type as x

angleInDegrees When it is true, the function will compute angle in degrees, otherwise they will be measured in radians

The function phase computes the rotation angle of each 2D vector that is formed from the corresponding elements of x and y:

$$\texttt{angle}(I) = \texttt{atan2}(\texttt{y}(I), \texttt{x}(I))$$

The angle estimation accuracy is $\sim 0.3^\circ$, when x(I)=y(I)=0, the corresponding angle(I) is set to 0.

See also:

cv::polarToCart

Computes x and y coordinates of 2D vectors from their magnitude and angle.

```
void polarToCart(const Mat& magnitude, const Mat& angle,
           Mat& x, Mat& y, bool angleInDegrees=false);
```

magnitude The source floating-point array of magnitudes of 2D vectors. It can be an empty matrix (=Mat()) - in this case the function assumes that all the magnitudes are =1. If it's not empty, it must have the same size and same type as angle

angle The source floating-point array of angles of the 2D vectors

x The destination array of x-coordinates of 2D vectors; will have the same size and the same type as angle

y The destination array of y-coordinates of 2D vectors; will have the same size and the same type as angle

angleInDegrees When it is true, the input angles are measured in degrees, otherwise they are measured in radians

The function polarToCart computes the cartesian coordinates of each 2D vector represented by the corresponding elements of magnitude and angle:

$$\begin{aligned}\mathtt{x}(I) &= \mathtt{magnitude}(I)\cos(\mathtt{angle}(I)) \\ \mathtt{y}(I) &= \mathtt{magnitude}(I)\sin(\mathtt{angle}(I))\end{aligned}$$

The relative accuracy of the estimated coordinates is $\sim 10^{-6}$.

See also: cv::cartToPolar, cv::magnitude, cv::phase, cv::exp, cv::log, cv::pow, cv::sqrt

cv::pow

Raises every array element to a power.

```
void pow(const Mat& src, double p, Mat& dst);
void pow(const MatND& src, double p, MatND& dst);
```

src The source array

p The exponent of power

dst The destination array; will have the same size and the same type as `src`

The function `pow` raises every element of the input array to `p`:

$$\texttt{dst}(I) = \begin{cases} \texttt{src}(I)^p & \text{if } \texttt{p} \text{ is integer} \\ |\texttt{src}(I)|^p & \text{otherwise} \end{cases}$$

That is, for a non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following example, computing the 5th root of array `src`, shows:

```
Mat mask = src < 0;
pow(src, 1./5, dst);
subtract(Scalar::all(0), dst, dst, mask);
```

For some values of `p`, such as integer values, 0.5, and -0.5, specialized faster algorithms are used.

See also: cv::sqrt, cv::exp, cv::log, cv::cartToPolar, cv::polarToCart

---

cv::randu

Generates a single uniformly-distributed random number or array of random numbers

```
template<typename _Tp> _Tp randu();
void randu(Mat& mtx, const Scalar& low, const Scalar& high);
```

mtx The output array of random numbers. The array must be pre-allocated and have 1 to 4 channels

low The inclusive lower boundary of the generated random numbers

high The exclusive upper boundary of the generated random numbers

The template functions `randu` generate and return the next uniformly-distributed random value of the specified type. `randu<int>()` is equivalent to `(int)theRNG();` etc. See cv::RNG description.

The second non-template variant of the function fills the matrix `mtx` with uniformly-distributed random numbers from the specified range:

$$\texttt{low}_c \leq \texttt{mtx}(I)_c < \texttt{high}_c$$

See also: cv::RNG, cv::randn, cv::theRNG.

---

cv::randn

Fills array with normally distributed random numbers

```
void randn(Mat& mtx, const Scalar& mean, const Scalar& stddev);
```

mtx The output array of random numbers. The array must be pre-allocated and have 1 to 4 channels

mean The mean value (expectation) of the generated random numbers

stddev The standard deviation of the generated random numbers

The function `randn` fills the matrix `mtx` with normally distributed random numbers with the specified mean and standard deviation. saturate_cast is applied to the generated numbers (i.e. the values are clipped)

See also: cv::RNG, cv::randu

---

cv::randShuffle

Shuffles the array elements randomly

```
void randShuffle(Mat& mtx, double iterFactor=1., RNG* rng=0);
```

mtx The input/output numerical 1D array

iterFactor The scale factor that determines the number of random swap operations. See the discussion

rng The optional random number generator used for shuffling. If it is zero, cv::theRNG() is used instead

The function `randShuffle` shuffles the specified 1D array by randomly choosing pairs of elements and swapping them. The number of such swap operations will be `mtx.rows*mtx.cols*iterFactor`

See also: cv::RNG, cv::sort

cv::reduce

Reduces a matrix to a vector

```
void reduce(const Mat& mtx, Mat& vec,
            int dim, int reduceOp, int dtype=-1);
```

mtx The source 2D matrix

vec The destination vector. Its size and type is defined by `dim` and `dtype` parameters

dim The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row and 1 means that the matrix is reduced to a single column

reduceOp The reduction operation, one of:

- **CV_REDUCE_SUM** The output is the sum of all of the matrix's rows/columns.
- **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
- **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/-columns.
- **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/-columns.

dtype When it is negative, the destination vector will have the same type as the source matrix, otherwise, its type will be `CV_MAKE_TYPE(CV_MAT_DEPTH(dtype), mtx.channels())`

The function `reduce` reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In the case of `CV_REDUCE_SUM` and `CV_REDUCE_AVG` the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

See also: cv::repeat

cv::repeat

Fill the destination array with repeated copies of the source array.

```
void repeat(const Mat& src, int ny, int nx, Mat& dst);
Mat repeat(const Mat& src, int ny, int nx);
```

src The source array to replicate

dst The destination array; will have the same type as `src`

ny How many times the `src` is repeated along the vertical axis

nx How many times the `src` is repeated along the horizontal axis

The functions cv::repeat duplicate the source array one or more times along each of the two axes:

$$\texttt{dst}_{ij} = \texttt{src}_{i \mod \texttt{src.rows}, j \mod \texttt{src.cols}}$$

The second variant of the function is more convenient to use with Matrix Expressions
See also: cv::reduce, Matrix Expressions

---

cv::saturate_cast

Template function for accurate conversion from one primitive type to another

```
template<typename _Tp> inline _Tp saturate_cast(unsigned char v);
template<typename _Tp> inline _Tp saturate_cast(signed char v);
template<typename _Tp> inline _Tp saturate_cast(unsigned short v);
template<typename _Tp> inline _Tp saturate_cast(signed short v);
template<typename _Tp> inline _Tp saturate_cast(int v);
template<typename _Tp> inline _Tp saturate_cast(unsigned int v);
template<typename _Tp> inline _Tp saturate_cast(float v);
template<typename _Tp> inline _Tp saturate_cast(double v);
```

v The function parameter

The functions `saturate_cast` resembles the standard C++ cast operations, such as `static_cast<T>()` etc. They perform an efficient and accurate conversion from one primitive type to another, see the introduction. "saturate" in the name means that when the input value `v` is out of range of the target type, the result will not be formed just by taking low bits of the input, but instead the value will be clipped. For example:

```
uchar a = saturate_cast<uchar>(-100); // a = 0 (UCHAR_MIN)
short b = saturate_cast<short>(33333.33333); // b = 32767 (SHRT_MAX)
```

Such clipping is done when the target type is `unsigned char, signed char, unsigned short or signed short` - for 32-bit integers no clipping is done.

When the parameter is floating-point value and the target type is an integer (8-, 16- or 32-bit), the floating-point value is first rounded to the nearest integer and then clipped if needed (when the target type is 8- or 16-bit).

This operation is used in most simple or complex image processing functions in OpenCV.

See also: cv::add, cv::subtract, cv::multiply, cv::divide, cv::Mat::convertTo

---

cv::scaleAdd

Calculates the sum of a scaled array and another array.

```
void scaleAdd(const Mat& src1, double scale,
              const Mat& src2, Mat& dst);
void scaleAdd(const MatND& src1, double scale,
              const MatND& src2, MatND& dst);
```

src1 The first source array

scale Scale factor for the first array

src2 The second source array; must have the same size and the same type as `src1`

dst The destination array; will have the same size and the same type as `src1`

The function `cvScaleAdd` is one of the classical primitive linear algebra operations, known as `DAXPY` or `SAXPY` in BLAS. It calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) + \texttt{src2}(I)$$

The function can also be emulated with a matrix expression, for example:

```
Mat A(3, 3, CV_64F);
...
A.row(0) = A.row(1)*2 + A.row(2);
```

See also: cv::add, cv::addWeighted, cv::subtract, cv::Mat::dot, cv::Mat::convertTo, Matrix Expressions

cv::setIdentity

Initializes a scaled identity matrix

```
void setIdentity(Mat& dst, const Scalar& value=Scalar(1));
```

dst The matrix to initialize (not necessarily square)

value The value to assign to the diagonal elements

The function cv::setIdentity initializes a scaled identity matrix:

$$\texttt{dst}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The function can also be emulated using the matrix initializers and the matrix expressions:

```
Mat A = Mat::eye(4, 3, CV_32F)*5;
// A will be set to [[5, 0, 0], [0, 5, 0], [0, 0, 5], [0, 0, 0]]
```

See also: cv::Mat::zeros, cv::Mat::ones, Matrix Expressions , cv::Mat::setTo, cv::Mat::operator=,

cv::solve

Solves one or more linear systems or least-squares problems.

```
bool solve(const Mat& src1, const Mat& src2,
           Mat& dst, int flags=DECOMP_LU);
```

src1 The input matrix on the left-hand side of the system

src2 The input matrix on the right-hand side of the system

dst The output solution

flags The solution (matrix inversion) method

DECOMP_LU Gaussian elimination with optimal pivot element chosen

DECOMP_CHOLESKY Cholesky LL^T factorization; the matrix `src1` must be symmetrical and positively defined

DECOMP_EIG Eigenvalue decomposition; the matrix `src1` must be symmetrical

DECOMP_SVD Singular value decomposition (SVD) method; the system can be over-defined and/or the matrix `src1` can be singular

DECOMP_QR QR factorization; the system can be over-defined and/or the matrix `src1` can be singular

DECOMP_NORMAL While all the previous flags are mutually exclusive, this flag can be used together with any of the previous. It means that the normal equations $\texttt{src1}^T \cdot \texttt{src1} \cdot \texttt{dst} = \texttt{src1}^T \texttt{src2}$ are solved instead of the original system $\texttt{src1} \cdot \texttt{dst} = \texttt{src2}$

The function `solve` solves a linear system or least-squares problem (the latter is possible with SVD or QR methods, or by specifying the flag `DECOMP_NORMAL`):

$$\texttt{dst} = \arg\min_X \|\texttt{src1} \cdot \texttt{X} - \texttt{src2}\|$$

If `DECOMP_LU` or `DECOMP_CHOLESKY` method is used, the function returns 1 if `src1` (or $\texttt{src1}^T\texttt{src1}$) is non-singular and 0 otherwise; in the latter case `dst` is not valid. Other methods find some pseudo-solution in the case of singular left-hand side part.

Note that if you want to find unity-norm solution of an under-defined singular system $\texttt{src1} \cdot \texttt{dst} = 0$, the function `solve` will not do the work. Use cv::SVD::solveZ instead.

See also: cv::invert, cv::SVD, cv::eigen

---

cv::solveCubic

Finds the real roots of a cubic equation.

```
void solveCubic(const Mat& coeffs, Mat& roots);
```

coeffs The equation coefficients, an array of 3 or 4 elements

roots The destination array of real roots which will have 1 or 3 elements

The function `solveCubic` finds the real roots of a cubic equation:
(if coeffs is a 4-element vector)

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or (if coeffs is 3-element vector):

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

The roots are stored to `roots` array.

cv::solvePoly

Finds the real or complex roots of a polynomial equation

```
void solvePoly(const Mat& coeffs, Mat& roots,
           int maxIters=20, int fig=100);
```

coeffs The array of polynomial coefficients

roots The destination (complex) array of roots

maxIters The maximum number of iterations the algorithm does

fig

The function `solvePoly` finds real and complex roots of a polynomial equation:

$$\texttt{coeffs}[0]x^n + \texttt{coeffs}[1]x^{n-1} + ... + \texttt{coeffs}[n-1]x + \texttt{coeffs}[n] = 0$$

cv::sort

Sorts each row or each column of a matrix

```
void sort(const Mat& src, Mat& dst, int flags);
```

src The source single-channel array

dst The destination array of the same size and the same type as `src`

flags The operation flags, a combination of the following values:

- **CV_SORT_EVERY_ROW** Each matrix row is sorted independently
- **CV_SORT_EVERY_COLUMN** Each matrix column is sorted independently. This flag and the previous one are mutually exclusive
- **CV_SORT_ASCENDING** Each matrix row is sorted in the ascending order
- **CV_SORT_DESCENDING** Each matrix row is sorted in the descending order. This flag and the previous one are also mutually exclusive

The function `sort` sorts each matrix row or each matrix column in ascending or descending order. If you want to sort matrix rows or columns lexicographically, you can use STL `std::sort` generic function with the proper comparison predicate.

See also: cv::sortIdx, cv::randShuffle

---

cv::sortIdx

Sorts each row or each column of a matrix

```
void sortIdx(const Mat& src, Mat& dst, int flags);
```

src The source single-channel array

dst The destination integer array of the same size as `src`

flags The operation flags, a combination of the following values:

- **CV_SORT_EVERY_ROW** Each matrix row is sorted independently
- **CV_SORT_EVERY_COLUMN** Each matrix column is sorted independently. This flag and the previous one are mutually exclusive
- **CV_SORT_ASCENDING** Each matrix row is sorted in the ascending order
- **CV_SORT_DESCENDING** Each matrix row is sorted in the descending order. This flag and the previous one are also mutually exclusive

The function `sortIdx` sorts each matrix row or each matrix column in ascending or descending order. Instead of reordering the elements themselves, it stores the indices of sorted elements in the destination array. For example:

```
Mat A = Mat::eye(3,3,CV_32F), B;
sortIdx(A, B, CV_SORT_EVERY_ROW + CV_SORT_ASCENDING);
// B will probably contain
// (because of equal elements in A some permutations are possible):
// [[1, 2, 0], [0, 2, 1], [0, 1, 2]]
```

See also: cv::sort, cv::randShuffle

---

cv::split

Divides multi-channel array into several single-channel arrays

```
void split(const Mat& mtx, Mat* mv);
void split(const Mat& mtx, vector<Mat>& mv);
void split(const MatND& mtx, MatND* mv);
void split(const MatND& mtx, vector<MatND>& mv);
```

mtx The source multi-channel array

mv The destination array or vector of arrays; The number of arrays must match `mtx.channels()`. The arrays themselves will be reallocated if needed

The functions `split` split multi-channel array into separate single-channel arrays:

$$\texttt{mv}[c](I) = \texttt{mtx}(I)_c$$

If you need to extract a single-channel or do some other sophisticated channel permutation, use cv::mixChannels

See also: cv::merge, cv::mixChannels, cv::cvtColor

---

cv::sqrt

Calculates square root of array elements

```
void sqrt(const Mat& src, Mat& dst);
void sqrt(const MatND& src, MatND& dst);
```

src The source floating-point array

dst The destination array; will have the same size and the same type as `src`

The functions `sqrt` calculate square root of each source array element. in the case of multi-channel arrays each channel is processed independently. The function accuracy is approximately the same as of the built-in `std::sqrt`.

See also: cv::pow, cv::magnitude

cv::subtract

Calculates per-element difference between two arrays or array and a scalar

```
void subtract(const Mat& src1, const Mat& src2, Mat& dst);
void subtract(const Mat& src1, const Mat& src2,
          Mat& dst, const Mat& mask);
void subtract(const Mat& src1, const Scalar& sc,
          Mat& dst, const Mat& mask=Mat());
void subtract(const Scalar& sc, const Mat& src2,
          Mat& dst, const Mat& mask=Mat());
void subtract(const MatND& src1, const MatND& src2, MatND& dst);
void subtract(const MatND& src1, const MatND& src2,
          MatND& dst, const MatND& mask);
void subtract(const MatND& src1, const Scalar& sc,
          MatND& dst, const MatND& mask=MatND());
void subtract(const Scalar& sc, const MatND& src2,
          MatND& dst, const MatND& mask=MatND());
```

src1 The first source array

src2 The second source array. It must have the same size and same type as `src1`

sc Scalar; the first or the second input parameter

dst The destination array; it will have the same size and same type as `src1`; see `Mat::create`

mask The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions `subtract` compute

- the difference between two arrays

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) - \texttt{src2}(I)) \quad \texttt{if mask}(I) \neq 0$$

- the difference between array and a scalar:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{src1}(I) - \texttt{sc}) \quad \texttt{if mask}(I) \neq 0$$

- the difference between scalar and an array:

$$\texttt{dst}(I) = \texttt{saturate}(\texttt{sc} - \texttt{src2}(I)) \quad \texttt{if mask}(I) \neq 0$$

where `I` is multi-dimensional index of array elements.
The first function in the above list can be replaced with matrix expressions:

```
dst = src1 - src2;
dst -= src2; // equivalent to subtract(dst, src2, dst);
```

See also: cv::add, cv::addWeighted, cv::scaleAdd, cv::convertScale, Matrix Expressions , saturate_cast.

SVD

Class for computing Singular Value Decomposition

```
class SVD
{
public:
    enum { MODIFY_A=1, NO_UV=2, FULL_UV=4 };newline
    // default empty constructor
    SVD();newline
    // decomposes m into u, w and vt: m = u*w*vt;newline
    // u and vt are orthogonal, w is diagonal
    SVD( const Mat& m, int flags=0 );newline
    // decomposes m into u, w and vt.
    SVD& operator ()( const Mat& m, int flags=0 );newline

    // finds such vector x, norm(x)=1, so that m*x = 0,
    // where m is singular matrix
    static void solveZ( const Mat& m, Mat& dst );newline
    // does back-subsitution:
    // dst = vt.t()*inv(w)*u.t()*rhs ~ inv(m)*rhs
    void backSubst( const Mat& rhs, Mat& dst ) const;newline

    Mat u, w, vt;
};
```

The class `SVD` is used to compute Singular Value Decomposition of a floating-point matrix and then use it to solve least-square problems, under-determined linear systems, invert matrices, compute condition numbers etc. For a bit faster operation you can pass `flags=SVD::MODIFY_A|...` to modify the decomposed matrix when it is not necessarily to preserve it. If you want to compute condition number of a matrix or absolute value of its determinant - you do not need `u` and `vt`, so

you can pass `flags=SVD::NO_UV|...`. Another flag `FULL_UV` indicates that full-size `u` and `vt` must be computed, which is not necessary most of the time.

See also: cv::invert, cv::solve, cv::eigen, cv::determinant

---

cv::sum

Calculates sum of array elements

```
Scalar sum(const Mat& mtx);
Scalar sum(const MatND& mtx);
```

mtx The source array; must have 1 to 4 channels

The functions `sum` calculate and return the sum of array elements, independently for each channel.

See also: cv::countNonZero, cv::mean, cv::meanStdDev, cv::norm, cv::minMaxLoc, cv::reduce

---

cv::theRNG

Returns the default random number generator

```
RNG& theRNG();
```

The function `theRNG` returns the default random number generator. For each thread there is separate random number generator, so you can use the function safely in multi-thread environments. If you just need to get a single random number using this generator or initialize an array, you can use cv::randu or cv::randn instead. But if you are going to generate many random numbers inside a loop, it will be much faster to use this function to retrieve the generator and then use `RNG::operator _Tp()`.

See also: cv::RNG, cv::randu, cv::randn

---

cv::trace

Returns the trace of a matrix

```
Scalar trace(const Mat& mtx);
```

mtx The source matrix

The function `trace` returns the sum of the diagonal elements of the matrix `mtx`.

$$\mathrm{tr}(\mathtt{mtx}) = \sum_i \mathtt{mtx}(i,i)$$

---

cv::transform

Performs matrix transformation of every array element.

```
void transform(const Mat& src,
               Mat& dst, const Mat& mtx );
```

src The source array; must have as many channels (1 to 4) as `mtx.cols` or `mtx.cols-1`

dst The destination array; will have the same size and depth as `src` and as many channels as `mtx.rows`

mtx The transformation matrix

The function `transform` performs matrix transformation of every element of array `src` and stores the results in `dst`:

$$\mathtt{dst}(I) = \mathtt{mtx} \cdot \mathtt{src}(I)$$

(when `mtx.cols=src.channels()`), or

$$\mathtt{dst}(I) = \mathtt{mtx} \cdot [\mathtt{src}(I); 1]$$

(when `mtx.cols=src.channels()+1`)

That is, every element of an `N`-channel array `src` is considered as `N`-element vector, which is transformed using a `M` × `N` or `M` × `N+1` matrix `mtx` into an element of `M`-channel array `dst`.

The function may be used for geometrical transformation of N-dimensional points, arbitrary linear color space transformation (such as various kinds of RGB→YUV transforms), shuffling the image channels and so forth.

See also: cv::perspectiveTransform, cv::getAffineTransform, cv::estimateRigidTransform, cv::warpAffine, cv::warpPerspective

cv::transpose

Transposes a matrix

```
void transpose(const Mat& src, Mat& dst);
```

src The source array

dst The destination array of the same type as `src`

The function cv::transpose transposes the matrix `src`:

$$\texttt{dst}(i,j) = \texttt{src}(j,i)$$

Note that no complex conjugation is done in the case of a complex matrix, it should be done separately if needed.

7.3 Dynamic Structures

7.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with `CV_RGB` or the Scalar constructor) for color images and brightness for grayscale images. For color images the order channel is normally *Blue, Green, Red*, this is what cv::imshow, cv::imread and cv::imwrite expect , so if you form a color using Scalar constructor, it should look like:

$$\texttt{Scalar}(blue\_component, green\_component, red\_component[, alpha\_component])$$

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using cv::cvtColor.

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as

`Point`$(x, y) \rightarrow$ `Point2f`$(x*2^{-shift}, y*2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

---

cv::circle

Draws a circle

```
void circle(Mat& img, Point center, int radius,
            const Scalar& color, int thickness=1,
            int lineType=8, int shift=0);
```

`img` Image where the circle is drawn

`center` Center of the circle

`radius` Radius of the circle

`color` Circle color

`thickness` Thickness of the circle outline if positive; negative thickness means that a filled circle is to be drawn

`lineType` Type of the circle boundary, see cv::line description

`shift` Number of fractional bits in the center coordinates and radius value

The function `circle` draws a simple or filled circle with a given center and radius.

---

cv::clipLine

Clips the line against the image rectangle

```
bool clipLine(Size imgSize, Point& pt1, Point& pt2);
bool clipLine(Rect imgRect, Point& pt1, Point& pt2);
```

imgSize The image size; the image rectangle will be `Rect(0, 0, imgSize.width, imgSize.height)`

imgSize The image rectangle

pt1 The first line point

pt2 The second line point

The functions `clipLine` calculate a part of the line segment which is entirely within the specified rectangle. They return `false` if the line segment is completely outside the rectangle and `true` otherwise.

---

cv::ellipse

Draws a simple or thick elliptic arc or an fills ellipse sector.

```
void ellipse(Mat& img, Point center, Size axes,
           double angle, double startAngle, double endAngle,
           const Scalar& color, int thickness=1,
           int lineType=8, int shift=0);
void ellipse(Mat& img, const RotatedRect& box, const Scalar& color,
           int thickness=1, int lineType=8);
```

img The image

center Center of the ellipse

axes Length of the ellipse axes

angle The ellipse rotation angle in degrees

startAngle Starting angle of the elliptic arc in degrees

endAngle Ending angle of the elliptic arc in degrees

box Alternative ellipse representation via a RotatedRect , i.e. the function draws an ellipse inscribed in the rotated rectangle

color Ellipse color

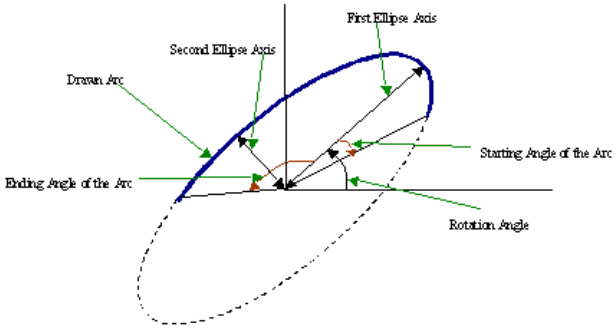
thickness Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn

lineType Type of the ellipse boundary, see cv::line description

shift Number of fractional bits in the center coordinates and axes' values

The functions `ellipse` with less parameters draw an ellipse outline, a filled ellipse, an elliptic arc or a filled ellipse sector. A piecewise-linear curve is used to approximate the elliptic arc boundary. If you need more control of the ellipse rendering, you can retrieve the curve using cv::ellipse2Poly and then render it with cv::polylines or fill it with cv::fillPoly. If you use the first variant of the function and want to draw the whole ellipse, not an arc, pass `startAngle=0` and `endAngle=360`. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



---

cv::ellipse2Poly

Approximates an elliptic arc with a polyline

```
void ellipse2Poly( Point center, Size axes, int angle,
           int startAngle, int endAngle, int delta,
           vector<Point>& pts );
```

center Center of the arc

axes Half-sizes of the arc. See cv::ellipse

angle Rotation angle of the ellipse in degrees. See cv::ellipse

startAngle Starting angle of the elliptic arc in degrees

endAngle Ending angle of the elliptic arc in degrees

delta Angle between the subsequent polyline vertices. It defines the approximation accuracy.

pts The output vector of polyline vertices

The function `ellipse2Poly` computes the vertices of a polyline that approximates the specified elliptic arc. It is used by cv::ellipse.

---

cv::fillConvexPoly

Fills a convex polygon.

```
void fillConvexPoly(Mat& img, const Point* pts, int npts,
           const Scalar& color, int lineType=8,
           int shift=0);
```

img Image

pts The polygon vertices

npts The number of polygon vertices

color Polygon color

lineType Type of the polygon boundaries, see cv::line description

shift The number of fractional bits in the vertex coordinates

The function `fillConvexPoly` draws a filled convex polygon. This function is much faster than the function `fillPoly` and can fill not only convex polygons but any monotonic polygon without self-intersections, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most (though, its top-most and/or the bottom edge could be horizontal).

---

cv::fillPoly

Fills the area bounded by one or more polygons

```
void fillPoly(Mat& img, const Point** pts,
           const int* npts, int ncontours,
           const Scalar& color, int lineType=8,
           int shift=0, Point offset=Point() );
```

img Image

pts Array of polygons, each represented as an array of points

npts The array of polygon vertex counters

ncontours The number of contours that bind the filled region

color Polygon color

lineType Type of the polygon boundaries, see cv::line description

shift The number of fractional bits in the vertex coordinates

The function `fillPoly` fills an area bounded by several polygonal contours. The function can fills complex areas, for example, areas with holes, contours with self-intersections (some of thier parts), and so forth.

---

cv::getTextSize

Calculates the width and height of a text string.

```
Size getTextSize(const string& text, int fontFace,
           double fontScale, int thickness,
           int* baseLine);
```

text The input text string

fontFace The font to use; see cv::putText

fontScale The font scale; see cv::putText

thickness The thickness of lines used to render the text; see cv::putText

baseLine The output parameter - y-coordinate of the baseline relative to the bottom-most text point

The function `getTextSize` calculates and returns size of the box that contain the specified text. That is, the following code will render some text, the tight box surrounding it and the baseline:

```
// Use "y" to show that the baseLine is about
string text = "Funny text inside the box";
int fontFace = FONT_HERSHEY_SCRIPT_SIMPLEX;
double fontScale = 2;
int thickness = 3;

Mat img(600, 800, CV_8UC3, Scalar::all(0));

int baseline=0;
Size textSize = getTextSize(text, fontFace,
                            fontScale, thickness, &baseline);
baseline += thickness;

// center the text
Point textOrg((img.cols - textSize.width)/2,
              (img.rows + textSize.height)/2);

// draw the box
rectangle(img, textOrg + Point(0, baseline),
          textOrg + Point(textSize.width, -textSize.height),
          Scalar(0,0,255));
// ... and the baseline first
line(img, textOrg + Point(0, thickness),
     textOrg + Point(textSize.width, thickness),
     Scalar(0, 0, 255));

// then put the text itself
putText(img, text, textOrg, fontFace, fontScale,
        Scalar::all(255), thickness, 8);
```

---

cv::line

Draws a line segment connecting two points

```
void line(Mat& img, Point pt1, Point pt2, const Scalar& color,
          int thickness=1, int lineType=8, int shift=0);
```

img The image

pt1 First point of the line segment

pt2 Second point of the line segment

color Line color

thickness Line thickness

lineType Type of the line:

- **8** (or omitted) 8-connected line.
- **4** 4-connected line.
- **CV_AA** antialiased line.

shift Number of fractional bits in the point coordinates

The function `line` draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image boundaries. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB(r, g, b)`.

LineIterator

Class for iterating pixels on a raster line

```
class LineIterator
{
public:
    // creates iterators for the line connecting pt1 and pt2
    // the line will be clipped on the image boundaries
    // the line is 8-connected or 4-connected
    // If leftToRight=true, then the iteration is always done
    // from the left-most point to the right most,
    // not to depend on the ordering of pt1 and pt2 parameters
    LineIterator(const Mat& img, Point pt1, Point pt2,
                 int connectivity=8, bool leftToRight=false);newline
    // returns pointer to the current line pixel
    uchar* operator *();newline
    // move the iterator to the next pixel
    LineIterator& operator ++();newline
    LineIterator operator ++(int);newline

    // internal state of the iterator
    uchar* ptr;newline
    int err, count;newline
```

```
    int minusDelta, plusDelta;newline
    int minusStep, plusStep;newline
};
```

The class `LineIterator` is used to get each pixel of a raster line. It can be treated as versatile implementation of the Bresenham algorithm, where you can stop at each pixel and do some extra processing, for example, grab pixel values along the line, or draw a line with some effect (e.g. with XOR operation).

The number of pixels along the line is store in `LineIterator::count`.

```
// grabs pixels along the line (pt1, pt2)
// from 8-bit 3-channel image to the buffer
LineIterator it(img, pt1, pt2, 8);
vector<Vec3b> buf(it.count);

for(int i = 0; i < it.count; i++, ++it)
    buf[i] = *(const Vec3b)*it;
```

---

cv::rectangle

Draws a simple, thick, or filled up-right rectangle.

```
void rectangle(Mat& img, Point pt1, Point pt2,
           const Scalar& color, int thickness=1,
           int lineType=8, int shift=0);
```

img Image

pt1 One of the rectangle's vertices

pt2 Opposite to `pt1` rectangle vertex

color Rectangle color or brightness (grayscale image)

thickness Thickness of lines that make up the rectangle. Negative values, e.g. `CV_FILLED`, mean that the function has to draw a filled rectangle.

lineType Type of the line, see cv::line description

shift Number of fractional bits in the point coordinates

The function `rectangle` draws a rectangle outline or a filled rectangle, which two opposite corners are `pt1` and `pt2`.

cv::polylines

Draws several polygonal curves

```
void polylines(Mat& img, const Point** pts, const int* npts,
           int ncontours, bool isClosed, const Scalar& color,
           int thickness=1, int lineType=8, int shift=0 );
```

img The image

pts Array of polygonal curves

npts Array of polygon vertex counters

ncontours The number of curves

isClosed Indicates whether the drawn polylines are closed or not. If they are closed, the function draws the line from the last vertex of each curve to its first vertex

color Polyline color

thickness Thickness of the polyline edges

lineType Type of the line segments, see cv::line description

shift The number of fractional bits in the vertex coordinates

The function `polylines` draws one or more polygonal curves.

cv::putText

Draws a text string

```
void putText( Mat& img, const string& text, Point org,
           int fontFace, double fontScale, Scalar color,
           int thickness=1, int lineType=8,
           bool bottomLeftOrigin=false );
```

img The image

text The text string to be drawn

org The bottom-left corner of the text string in the image

fontFace The font type, one of FONT_HERSHEY_SIMPLEX, FONT_HERSHEY_PLAIN, FONT_HERSHEY_DUPLEX, FONT_HERSHEY_COMPLEX, FONT_HERSHEY_TRIPLEX, FONT_HERSHEY_COMPLEX_SMALL, FONT_HERSHEY_S or FONT_HERSHEY_SCRIPT_COMPLEX, where each of the font id's can be combined with FONT_HERSHEY_ITALIC to get the slanted letters.

fontScale The font scale factor that is multiplied by the font-specific base size

thickness Thickness of the lines used to draw the text

lineType The line type; see line for details

bottomLeftOrigin When true, the image data origin is at the bottom-left corner, otherwise it's at the top-left corner

The function putText draws a text string in the image. Symbols that can not be rendered using the specified font are replaced question marks. See cv::getTextSize for a text rendering code example.

7.5 XML/YAML Persistence

FileStorage

The XML/YAML file storage class

```
class FileStorage
{
public:
    enum { READ=0, WRITE=1, APPEND=2 };
    enum { UNDEFINED=0, VALUE_EXPECTED=1, NAME_EXPECTED=2, INSIDE_MAP=4 };
    // the default constructor
    FileStorage();
    // the constructor that opens the file for reading
    // (flags=FileStorage::READ) or writing (flags=FileStorage::WRITE)
    FileStorage(const string& filename, int flags);
    // wraps the already opened CvFileStorage*
    FileStorage(CvFileStorage* fs);
    // the destructor; closes the file if needed
    virtual ~FileStorage();
```

```
    // opens the specified file for reading (flags=FileStorage::READ)
    // or writing (flags=FileStorage::WRITE)
    virtual bool open(const string& filename, int flags);
    // checks if the storage is opened
    virtual bool isOpened() const;
    // closes the file
    virtual void release();

    // returns the first top-level node
    FileNode getFirstTopLevelNode() const;
    // returns the root file node
    // (it's the parent of the first top-level node)
    FileNode root(int streamidx=0) const;
    // returns the top-level node by name
    FileNode operator[](const string& nodename) const;
    FileNode operator[](const char* nodename) const;

    // returns the underlying CvFileStorage*
    CvFileStorage* operator *() { return fs; }
    const CvFileStorage* operator *() const { return fs; }

    // writes the certain number of elements of the specified format
    // (see DataType) without any headers
    void writeRaw( const string& fmt, const uchar* vec, size_t len );

    // writes an old-style object (CvMat, CvMatND etc.)
    void writeObj( const string& name, const void* obj );

    // returns the default object name from the filename
    // (used by cvSave() with the default object name etc.)
    static string getDefaultObjectName(const string& filename);

    Ptr<CvFileStorage> fs;
    string elname;
    vector<char> structs;
    int state;
};
```

FileNode

The XML/YAML file node class

```
class CV_EXPORTS FileNode
```

```
{
public:
    enum { NONE=0, INT=1, REAL=2, FLOAT=REAL, STR=3,
        STRING=STR, REF=4, SEQ=5, MAP=6, TYPE_MASK=7,
        FLOW=8, USER=16, EMPTY=32, NAMED=64 };
    FileNode();
    FileNode(const CvFileStorage* fs, const CvFileNode* node);
    FileNode(const FileNode& node);
    FileNode operator[](const string& nodename) const;
    FileNode operator[](const char* nodename) const;
    FileNode operator[](int i) const;
    int type() const;
    int rawDataSize(const string& fmt) const;
    bool empty() const;
    bool isNone() const;
    bool isSeq() const;
    bool isMap() const;
    bool isInt() const;
    bool isReal() const;
    bool isString() const;
    bool isNamed() const;
    string name() const;
    size_t size() const;
    operator int() const;
    operator float() const;
    operator double() const;
    operator string() const;

    FileNodeIterator begin() const;
    FileNodeIterator end() const;

    void readRaw( const string& fmt, uchar* vec, size_t len ) const;
    void* readObj() const;

    // do not use wrapper pointer classes for better efficiency
    const CvFileStorage* fs;
    const CvFileNode* node;
};
```

FileNodeIterator

The XML/YAML file node iterator class

```
class CV_EXPORTS FileNodeIterator
```

```
{
public:
    FileNodeIterator();
    FileNodeIterator(const CvFileStorage* fs,
        const CvFileNode* node, size_t ofs=0);
    FileNodeIterator(const FileNodeIterator& it);
    FileNode operator *() const;
    FileNode operator ->() const;

    FileNodeIterator& operator ++();
    FileNodeIterator operator ++(int);
    FileNodeIterator& operator --();
    FileNodeIterator operator --(int);
    FileNodeIterator& operator += (int);
    FileNodeIterator& operator -= (int);

    FileNodeIterator& readRaw( const string& fmt, uchar* vec,
                               size_t maxCount=(size_t)INT_MAX );

    const CvFileStorage* fs;
    const CvFileNode* container;
    CvSeqReader reader;
    size_t remaining;
};
```

7.6 Clustering and Search in Multi-Dimensional Spaces

---

cv::kmeans

Finds the centers of clusters and groups the input samples around the clusters.

```
double kmeans( const Mat& samples, int clusterCount, Mat& labels,
           TermCriteria termcrit, int attempts,
           int flags, Mat* centers );
```

samples Floating-point matrix of input samples, one row per sample

clusterCount The number of clusters to split the set by

labels The input/output integer array that will store the cluster indices for every sample

termcrit Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

attempts How many times the algorithm is executed using different initial labelings. The algorithm returns the labels that yield the best compactness (see the last function parameter)

flags It can take the following values:

KMEANS_RANDOM_CENTERS Random initial centers are selected in each attempt

KMEANS_PP_CENTERS Use kmeans++ center initialization by Arthur and Vassilvitskii

KMEANS_USE_INITIAL_LABELS During the first (and possibly the only) attempt, the function uses the user-supplied labels instaed of computing them from the initial centers. For the second and further attempts, the function will use the random or semi-random centers (use one of `KMEANS_*_CENTERS` flag to specify the exact method)

centers The output matrix of the cluster centers, one row per each cluster center

The function `kmeans` implements a k-means algorithm that finds the centers of `clusterCount` clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a 0-based cluster index for the sample stored in the i^{th} row of the `samples` matrix.

The function returns the compactness measure, which is computed as

$$\sum_i \|\texttt{samples}_i - \texttt{centers}_{\texttt{labels}_i}\|^2$$

after every attempt; the best (minimum) value is chosen and the corresponding labels and the compactness value are returned by the function. Basically, the user can use only the core of the function, set the number of attempts to 1, initialize labels each time using some custom algorithm and pass them with

(`flags=KMEANS_USE_INITIAL_LABELS`) flag, and then choose the best (most-compact) clustering.

---

cv::partition

Splits an element set into equivalency classes.

```
template<typename _Tp, class _EqPredicate> int
partition( const vector<_Tp>& vec, vector<int>& labels,
           _EqPredicate predicate=_EqPredicate());
```

vec The set of elements stored as a vector

labels The output vector of labels; will contain as many elements as `vec`. Each label `labels[i]` is 0-based cluster index of `vec[i]`

predicate The equivalence predicate (i.e. pointer to a boolean function of two arguments or an instance of the class that has the method `bool operator()(const _Tp& a, const _Tp& b)`. The predicate returns true when the elements are certainly if the same class, and false if they may or may not be in the same class

The generic function `partition` implements an $O(N^2)$ algorithm for splitting a set of N elements into one or more equivalency classes, as described in http://en.wikipedia.org/wiki/Disjoint-set_data_structure. The function returns the number of equivalency classes.

Fast Approximate Nearest Neighbor Search

This section documents OpenCV's interface to the FLANN[1] library. FLANN (Fast Library for Approximate Nearest Neighbors) is a library that contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. More information about FLANN can be found in [**?**].

---

flann::Index

The FLANN nearest neighbor index class.

```
namespace flann
{
    class Index
    {
    public:
            Index(const Mat& features, const IndexParams& params);

            void knnSearch(const vector<float>& query,
                           vector<int>& indices,
                           vector<float>& dists,
                           int knn,
                           const SearchParams& params);
            void knnSearch(const Mat& queries,
                           Mat& indices,
                           Mat& dists,
                           int knn,
```

[1] http://people.cs.ubc.ca/~mariusm/flann

```
                            const SearchParams& params);

            int radiusSearch(const vector<float>& query,
                             vector<int>& indices,
                             vector<float>& dists,
                             float radius,
                             const SearchParams& params);
            int radiusSearch(const Mat& query,
                             Mat& indices,
                             Mat& dists,
                             float radius,
                             const SearchParams& params);

            void save(std::string filename);

            int veclen() const;

            int size() const;
    };
}
```

---

cv::flann::Index::Index

Constructs a nearest neighbor search index for a given dataset.

```
Index::Index(const Mat& features, const IndexParams& params);
```

features Matrix of type CV_32F containing the features(points) to index. The size of the matrix is num_features x feature_dimensionality.

params Structure containing the index parameters. The type of index that will be constructed depends on the type of this parameter. The possible parameter types are:

LinearIndexParams When passing an object of this type, the index will perform a linear, brute-force search.

```
        struct LinearIndexParams : public IndexParams
        {
        };
```

KDTreeIndexParams When passing an object of this type the index constructed will consist of a set of randomized kd-trees which will be searched in parallel.

```
struct KDTreeIndexParams : public IndexParams
{
KDTreeIndexParams( int trees = 4 );
};
```

trees The number of parallel kd-trees to use. Good values are in the range [1..16]

KMeansIndexParams When passing an object of this type the index constructed will be a hierarchical k-means tree.

```
struct KMeansIndexParams : public IndexParams
{
KMeansIndexParams( int branching = 32,
           int iterations = 11,
           flann_centers_init_t centers_init = CENTERS_RANDOM,
           float cb_index = 0.2 );
};
```

branching The branching factor to use for the hierarchical k-means tree

iterations The maximum number of iterations to use in the k-means clustering stage when building the k-means tree. A value of -1 used here means that the k-means clustering should be iterated until convergence

centers_init The algorithm to use for selecting the initial centers when performing a k-means clustering step. The possible values are CENTERS_RANDOM (picks the initial cluster centers randomly), CENTERS_GONZALES (picks the initial centers using Gonzales' algorithm) and CENTERS_KMEANSPP (picks the initial centers using the algorithm suggested in [**?**])

cb_index This parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When `cb_index` is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

CompositeIndexParams When using a parameters object of this type the index created combines the randomized kd-trees and the hierarchical k-means tree.

```
struct CompositeIndexParams : public IndexParams
{
CompositeIndexParams( int trees = 4,
           int branching = 32,
           int iterations = 11,
           flann_centers_init_t centers_init = CENTERS_RANDOM,
           float cb_index = 0.2 );
};
```

AutotunedIndexParams When passing an object of this type the index created is automatically tuned to offer the best performance, by choosing the optimal index type (randomized kd-trees, hierarchical kmeans, linear) and parameters for the dataset provided.

```
struct AutotunedIndexParams : public IndexParams
{
AutotunedIndexParams( float target_precision = 0.9,
           float build_weight = 0.01,
           float memory_weight = 0,
           float sample_fraction = 0.1 );
};
```

target_precision Is a number between 0 and 1 specifying the percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

build_weight Specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times.

memory_weight Is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

sample_fraction Is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm

on the full dataset gives the most accurate results, but for very large datasets can take longer than desired. In such case using just a fraction of the data helps speeding up this algorithm while still giving good approximations of the optimum parameters.

SavedIndexParams This object type is used for loading a previously saved index from the disk.

```
struct SavedIndexParams : public IndexParams
{
SavedIndexParams( std::string filename );
};
```

filename The filename in which the index was saved.

---

cv::flann::Index::knnSearch

Performs a K-nearest neighbor search for a given query point using the index.

```
void Index::knnSearch(const vector<float>& query,
           vector<int>& indices,
           vector<float>& dists,
           int knn,
           const SearchParams& params);
```

query The query point

indices Vector that will contain the indices of the K-nearest neighbors found. It must have at least knn size.

dists Vector that will contain the distances to the K-nearest neighbors found. It must have at least knn size.

knn Number of nearest neighbors to search for.

params Search parameters

```
struct SearchParams {
        SearchParams(int checks = 32);
};
```

checks The number of times the tree(s) in the index should be recursively traversed. A higher value for this parameter would give better search precision, but also take more time. If automatic configuration was used when the index was created, the number of checks required to achieve the specified precision was also computed, in which case this parameter is ignored.

---

cv::flann::Index::knnSearch

Performs a K-nearest neighbor search for multiple query points.

```
void Index::knnSearch(const Mat& queries,
           Mat& indices, Mat& dists,
           int knn, const SearchParams& params);
```

queries The query points, one per row

indices Indices of the nearest neighbors found

dists Distances to the nearest neighbors found

knn Number of nearest neighbors to search for

params Search parameters

---

cv::flann::Index::radiusSearch

Performs a radius nearest neighbor search for a given query point.

```
int Index::radiusSearch(const vector<float>& query,
           vector<int>& indices,
           vector<float>& dists,
           float radius,
           const SearchParams& params);
```

query The query point

indices Vector that will contain the indices of the points found within the search radius in decreasing order of the distance to the query point. If the number of neighbors in the search radius is bigger than the size of this vector, the ones that don't fit in the vector are ignored.

dists Vector that will contain the distances to the points found within the search radius

radius The search radius

params Search parameters

---

cv::flann::Index::radiusSearch

Performs a radius nearest neighbor search for multiple query points.

```
int Index::radiusSearch(const Mat& query,
           Mat& indices,
           Mat& dists,
           float radius,
           const SearchParams& params);
```

queries The query points, one per row

indices Indices of the nearest neighbors found

dists Distances to the nearest neighbors found

radius The search radius

params Search parameters

---

cv::flann::Index::save

Saves the index to a file.

```
void Index::save(std::string filename);
```

filename The file to save the index to

cv::flann::hierarchicalClustering

Clusters the given points by constructing a hierarchical k-means tree and choosing a cut in the tree that minimizes the cluster's variance.

```
int hierarchicalClustering(const Mat& features, Mat& centers,
           const KMeansIndexParams& params);
```

features The points to be clustered

centers The centers of the clusters obtained. The number of rows in this matrix represents the number of clusters desired, however, because of the way the cut in the hierarchical tree is choosen, the number of clusters computed will be the highest number of the form $(branching - 1) * k + 1$ that's lower than the number of clusters desired, where $branching$ is the tree's branching factor (see description of the KMeansIndexParams).

params Parameters used in the construction of the hierarchical k-means tree

The function returns the number of clusters computed.

7.7 Utility and System Functions and Macros

cv::alignPtr

Aligns pointer to the specified number of bytes

```
template<typename _Tp> _Tp* alignPtr(_Tp* ptr, int n=sizeof(_Tp));
```

ptr The aligned pointer

n The alignment size; must be a power of two

The function returns the aligned pointer of the same type as the input pointer:

```
(_Tp*)(((size_t)ptr + n-1) & -n)
```

cv::alignSize

Aligns a buffer size to the specified number of bytes

```
size_t alignSize(size_t sz, int n);
```

sz The buffer size to align

n The alignment size; must be a power of two

The function returns the minimum number that is greater or equal to `sz` and is divisble by `n`:

```
(sz + n-1) & -n
```

cv::allocate

Allocates an array of elements

```
template<typename _Tp> _Tp* allocate(size_t n);
```

n The number of elements to allocate

The generic function `allocate` allocates buffer for the specified number of elements. For each element the default constructor is called.

cv::deallocate

Allocates an array of elements

```
template<typename _Tp> void deallocate(_Tp* ptr, size_t n);
```

ptr Pointer to the deallocated buffer

n The number of elements in the buffer

The generic function `deallocate` deallocates the buffer allocated with cv::allocate. The number of elements must match the number passed to cv::allocate.

CV_Assert

Checks a condition at runtime.

```
CV_Assert(expr)
```

```
#define CV_Assert( expr ) ...
#define CV_DbgAssert(expr) ...
```

expr The checked expression

The macros `CV_Assert` and `CV_DbgAssert` evaluate the specified expression and if it is 0, the macros raise an error (see cv::error). The macro `CV_Assert` checks the condition in both Debug and Release configurations, while `CV_DbgAssert` is only retained in the Debug configuration.

cv::error

Signals an error and raises the exception

```
void error( const Exception& exc );
#define CV_Error( code, msg ) <...>
#define CV_Error_( code, args ) <...>
```

exc The exception to throw

code The error code, normally, a negative value. The list of pre-defined error codes can be found in `cxerror.h`

msg Text of the error message

args printf-like formatted error message in parantheses

The function and the helper macros `CV_Error` and `CV_Error_` call the error handler. Currently, the error handler prints the error code (`exc.code`), the context (`exc.file`, `exc.line` and the error message `exc.err` to the standard error stream `stderr`. In Debug configuration it then provokes memory access violation, so that the execution stack and all the parameters can be analyzed in debugger. In Release configuration the exception `exc` is thrown.

The macro `CV_Error_` can be used to construct the error message on-fly to include some dynamic information, for example:

```
// note the extra parentheses around the formatted text message
CV_Error_(CV_StsOutOfRange,
    ("the matrix element (%d,%d)=%g is out of range",
    i, j, mtx.at<float>(i,j)))
```

Exception

The exception class passed to error

```
class Exception
{
public:
    // various constructors and the copy operation
    Exception() { code = 0; line = 0; }
    Exception(int _code, const string& _err,
              const string& _func, const string& _file, int _line);newline
    Exception(const Exception& exc);newline
    Exception& operator = (const Exception& exc);newline

    // the error code
    int code;newline
    // the error text message
    string err;newline
    // function name where the error happened
    string func;newline
    // the source file name where the error happened
    string file;newline
    // the source file line where the error happened
    int line;
};
```

The class `Exception` encapsulates all or almost all the necessary information about the error happened in the program. The exception is usually constructed and thrown implicitly, via `CV_Error` and `CV_Error_` macros, see cv::error.

cv::fastMalloc

Allocates aligned memory buffer

```
void* fastMalloc(size_t size);
```

size The allocated buffer size

The function allocates buffer of the specified size and returns it. When the buffer size is 16 bytes or more, the returned buffer is aligned on 16 bytes.

---

cv::fastFree

Deallocates memory buffer

```
void fastFree(void* ptr);
```

ptr Pointer to the allocated buffer

The function deallocates the buffer, allocated with cv::fastMalloc. If NULL pointer is passed, the function does nothing.

---

cv::format

Returns a text string formatted using printf-like expression

```
string format( const char* fmt, ... );
```

fmt The printf-compatible formatting specifiers

The function acts like `sprintf`, but forms and returns STL string. It can be used for form the error message in cv::Exception constructor.

---

cv::getNumThreads

Returns the number of threads used by OpenCV

```
int getNumThreads();
```

The function returns the number of threads that is used by OpenCV.
See also: cv::setNumThreads, cv::getThreadNum.

cv::getThreadNum

Returns index of the currently executed thread

```
int getThreadNum();
```

The function returns 0-based index of the currently executed thread. The function is only valid inside a parallel OpenMP region. When OpenCV is built without OpenMP support, the function always returns 0.

See also: cv::setNumThreads, cv::getNumThreads.

cv::getTickCount

Returns the number of ticks

```
int64 getTickCount();
```

The function returns the number of ticks since the certain event (e.g. when the machine was turned on). It can be used to initialize cv::RNG or to measure a function execution time by reading the tick count before and after the function call. See also the tick frequency.

cv::getTickFrequency

Returns the number of ticks per second

```
double getTickFrequency();
```

The function returns the number of ticks per second. That is, the following code computes the executing time in seconds.

```
double t = (double)getTickCount();
// do something ...
t = ((double)getTickCount() - t)/getTickFrequency();
```

cv::setNumThreads

Sets the number of threads used by OpenCV

```
void setNumThreads(int nthreads);
```

nthreads The number of threads used by OpenCV

The function sets the number of threads used by OpenCV in parallel OpenMP regions. If `nthreads=0`, the function will use the default number of threads, which is usually equal to the number of the processing cores.

See also: cv::getNumThreads, cv::getThreadNum

Chapter 8

cv. Image Processing and Computer Vision

8.1 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as cv::Mat's), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y). It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc. OpenCV let the user to specify the extrapolation method; see the function cv::borderInterpolate and discussion of `borderType` parameter in various functions below.

BaseColumnFilter

Base class for filters with single-column kernels

```
class BaseColumnFilter
```

```
{
public:
    virtual ~BaseColumnFilter();

    // To be overriden by the user.
    //
    // runs filtering operation on the set of rows,
    // "dstcount + ksize - 1" rows on input,
    // "dstcount" rows on output,
    // each input and output row has "width" elements
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, int dststep,
                            int dstcount, int width) = 0;
    // resets the filter state (may be needed for IIR filters)
    virtual void reset();

    int ksize; // the aperture size
    int anchor; // position of the anchor point,
                // normally not used during the processing
};
```

The class `BaseColumnFilter` is the base class for filtering data using single-column kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\texttt{dst}(x,y) = F(\texttt{src}[y](x),\ \texttt{src}[y+1](x),\ ...,\ \texttt{src}[y+\texttt{ksize}-1](x)$$

where F is the filtering function, but, as it is represented as a class, it can produce any side effects, memorize previously processed data etc. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to cv::FilterEngine constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: cv::BaseRowFilter, cv::BaseFilter, cv::FilterEngine, cv::getColumnSumFilter, cv::getLinearColumnFilter, cv::getMorphologyColumnFilter

---

BaseFilter

Base class for 2D image filters

```
class BaseFilter
{
public:
    virtual ~BaseFilter();
```

```
    // To be overriden by the user.
    //
    // runs filtering operation on the set of rows,
    // "dstcount + ksize.height - 1" rows on input,
    // "dstcount" rows on output,
    // each input row has "(width + ksize.width-1)*cn" elements
    // each output row has "width*cn" elements.
    // the filtered rows are written into "dst" buffer.
    virtual void operator()(const uchar** src, uchar* dst, int dststep,
                            int dstcount, int width, int cn) = 0;
    // resets the filter state (may be needed for IIR filters)
    virtual void reset();
    Size ksize;
    Point anchor;
};
```

The class `BaseFilter` is the base class for filtering data using 2D kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\begin{aligned}&\texttt{dst}(x,y) = F(\texttt{src}[y](x),\ \texttt{src}[y](x+1),\ ...,\ \texttt{src}[y](x+\texttt{ksize.width}-1),\\&\texttt{src}[y+1](x),\ \texttt{src}[y+1](x+1),\ ...,\ \texttt{src}[y+1](x+\texttt{ksize.width}-1),\\&.....\\&\texttt{src}[y+\texttt{ksize.height}-1](x),\\&\texttt{src}[y+\texttt{ksize.height}-1](x+1),\\&...\texttt{src}[y+\texttt{ksize.height}-1](x+\texttt{ksize.width}-1))\end{aligned}$$

where F is the filtering function. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to cv::FilterEngine constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: cv::BaseColumnFilter, cv::BaseRowFilter, cv::FilterEngine, cv::getLinearFilter, cv::getMorphologyFilter

---

BaseRowFilter

Base class for filters with single-row kernels

```
class BaseRowFilter
{
public:
    virtual ~BaseRowFilter();

    // To be overriden by the user.
```

```
    //
    // runs filtering operation on the single input row
    // of "width" element, each element is has "cn" channels.
    // the filtered row is written into "dst" buffer.
    virtual void operator()(const uchar* src, uchar* dst,
                            int width, int cn) = 0;
    int ksize, anchor;
};
```

The class `BaseRowFilter` is the base class for filtering data using single-row kernels. The filtering does not have to be a linear operation. In general, it could be written as following:

$$\texttt{dst}(x,y) = F(\texttt{src}[y](x),\ \texttt{src}[y](x+1),\ ...,\ \texttt{src}[y](x+\texttt{ksize.width}-1))$$

where F is the filtering function. The class only defines the interface and is not used directly. Instead, there are several functions in OpenCV (and you can add more) that return pointers to the derived classes that implement specific filtering operations. Those pointers are then passed to cv::FilterEngine constructor. While the filtering operation interface uses `uchar` type, a particular implementation is not limited to 8-bit data.

See also: cv::BaseColumnFilter, cv::Filter, cv::FilterEngine, cv::getLinearRowFilter, cv::getMorphologyRow cv::getRowSumFilter

---

FilterEngine

Generic image filtering class

```
class FilterEngine
{
public:
    // empty constructor
    FilterEngine();
    // builds a 2D non-separable filter (!_filter2D.empty()) or
    // a separable filter (!_rowFilter.empty() && !_columnFilter.empty())
    // the input data type will be "srcType", the output data type will be "dstType",
    // the intermediate data type is "bufType".
    // _rowBorderType and _columnBorderType determine how the image
    // will be extrapolated beyond the image boundaries.
    // _borderValue is only used when _rowBorderType and/or _columnBorderType
    // == cv::BORDER_CONSTANT
    FilterEngine(const Ptr<BaseFilter>& _filter2D,
                 const Ptr<BaseRowFilter>& _rowFilter,
                 const Ptr<BaseColumnFilter>& _columnFilter,
                 int srcType, int dstType, int bufType,
                 int _rowBorderType=BORDER_REPLICATE,
```

```
                int _columnBorderType=-1, // use _rowBorderType by default
                const Scalar& _borderValue=Scalar());
    virtual ~FilterEngine();
    // separate function for the engine initialization
    void init(const Ptr<BaseFilter>& _filter2D,
              const Ptr<BaseRowFilter>& _rowFilter,
              const Ptr<BaseColumnFilter>& _columnFilter,
              int srcType, int dstType, int bufType,
              int _rowBorderType=BORDER_REPLICATE, int _columnBorderType=-1,
              const Scalar& _borderValue=Scalar());
    // starts filtering of the ROI in an image of size "wholeSize".
    // returns the starting y-position in the source image.
    virtual int start(Size wholeSize, Rect roi, int maxBufRows=-1);
    // alternative form of start that takes the image
    // itself instead of "wholeSize". Set isolated to true to pretend that
    // there are no real pixels outside of the ROI
    // (so that the pixels will be extrapolated using the specified border modes)
    virtual int start(const Mat& src, const Rect& srcRoi=Rect(0,0,-1,-1),
                      bool isolated=false, int maxBufRows=-1);
    // processes the next portion of the source image,
    // "srcCount" rows starting from "src" and
    // stores the results to "dst".
    // returns the number of produced rows
    virtual int proceed(const uchar* src, int srcStep, int srcCount,
                        uchar* dst, int dstStep);
    // higher-level function that processes the whole
    // ROI or the whole image with a single call
    virtual void apply( const Mat& src, Mat& dst,
                        const Rect& srcRoi=Rect(0,0,-1,-1),
                        Point dstOfs=Point(0,0),
                        bool isolated=false);
    bool isSeparable() const { return filter2D.empty(); }
    // how many rows from the input image are not yet processed
    int remainingInputRows() const;
    // how many output rows are not yet produced
    int remainingOutputRows() const;
    ...
    // the starting and the ending rows in the source image
    int startY, endY;

    // pointers to the filters
    Ptr<BaseFilter> filter2D;
    Ptr<BaseRowFilter> rowFilter;
    Ptr<BaseColumnFilter> columnFilter;
};
```

The class `FilterEngine` can be used to apply an arbitrary filtering operation to an image. It contains all the necessary intermediate buffers, it computes extrapolated values of the ”virtual” pixels outside of the image etc. Pointers to the initialized `FilterEngine` instances are returned by various `create*Filter` functions, see below, and they are used inside high-level functions such as cv::filter2D, cv::erode, cv::dilate etc, that is, the class is the workhorse in many of OpenCV filtering functions.

This class makes it easier (though, maybe not very easy yet) to combine filtering operations with other operations, such as color space conversions, thresholding, arithmetic operations, etc. By combining several operations together you can get much better performance because your data will stay in cache. For example, below is the implementation of Laplace operator for a floating-point images, which is a simplified implementation of cv::Laplacian:

```
void laplace_f(const Mat& src, Mat& dst)
{
    CV_Assert( src.type() == CV_32F );
    dst.create(src.size(), src.type());

    // get the derivative and smooth kernels for d2I/dx2.
    // for d2I/dy2 we could use the same kernels, just swapped
    Mat kd, ks;
    getSobelKernels( kd, ks, 2, 0, ksize, false, ktype );

    // let's process 10 source rows at once
    int DELTA = std::min(10, src.rows);
    Ptr<FilterEngine> Fxx = createSeparableLinearFilter(src.type(),
        dst.type(), kd, ks, Point(-1,-1), 0, borderType, borderType, Scalar() );
    Ptr<FilterEngine> Fyy = createSeparableLinearFilter(src.type(),
        dst.type(), ks, kd, Point(-1,-1), 0, borderType, borderType, Scalar() );

    int y = Fxx->start(src), dsty = 0, dy = 0;
    Fyy->start(src);
    const uchar* sptr = src.data + y*src.step;

    // allocate the buffers for the spatial image derivatives;
    // the buffers need to have more than DELTA rows, because at the
    // last iteration the output may take max(kd.rows-1,ks.rows-1)
    // rows more than the input.
    Mat Ixx( DELTA + kd.rows - 1, src.cols, dst.type() );
    Mat Iyy( DELTA + kd.rows - 1, src.cols, dst.type() );

    // inside the loop we always pass DELTA rows to the filter
    // (note that the "proceed" method takes care of possibe overflow, since
    // it was given the actual image height in the "start" method)
    // on output we can get:
```

```
    //  * < DELTA rows (the initial buffer accumulation stage)
    //  * = DELTA rows (settled state in the middle)
    //  * > DELTA rows (then the input image is over, but we generate
    //                  "virtual" rows using the border mode and filter them)
    // this variable number of output rows is dy.
    // dsty is the current output row.
    // sptr is the pointer to the first input row in the portion to process
    for( ; dsty < dst.rows; sptr += DELTA*src.step, dsty += dy )
    {
        Fxx->proceed( sptr, (int)src.step, DELTA, Ixx.data, (int)Ixx.step );
        dy = Fyy->proceed( sptr, (int)src.step, DELTA, d2y.data, (int)Iyy.step );
        if( dy > 0 )
        {
            Mat dstripe = dst.rowRange(dsty, dsty + dy);
            add(Ixx.rowRange(0, dy), Iyy.rowRange(0, dy), dstripe);
        }
    }
}
```

If you do not need that much control of the filtering process, you can simply use the `FilterEngine::apply` method. Here is how the method is actually implemented:

```
void FilterEngine::apply(const Mat& src, Mat& dst,
    const Rect& srcRoi, Point dstOfs, bool isolated)
{
    // check matrix types
    CV_Assert( src.type() == srcType && dst.type() == dstType );

    // handle the "whole image" case
    Rect _srcRoi = srcRoi;
    if( _srcRoi == Rect(0,0,-1,-1) )
        _srcRoi = Rect(0,0,src.cols,src.rows);

    // check if the destination ROI is inside the dst.
    // and FilterEngine::start will check if the source ROI is inside src.
    CV_Assert( dstOfs.x >= 0 && dstOfs.y >= 0 &&
        dstOfs.x + _srcRoi.width <= dst.cols &&
        dstOfs.y + _srcRoi.height <= dst.rows );

    // start filtering
    int y = start(src, _srcRoi, isolated);

    // process the whole ROI. Note that "endY - startY" is the total number
    // of the source rows to process
    // (including the possible rows outside of srcRoi but inside the source image)
    proceed( src.data + y*src.step,
```

```
            (int)src.step, endY - startY,
            dst.data + dstOfs.y*dst.step +
            dstOfs.x*dst.elemSize(), (int)dst.step );
}
```

Unlike the earlier versions of OpenCV, now the filtering operations fully support the notion of image ROI, that is, pixels outside of the ROI but inside the image can be used in the filtering operations. For example, you can take a ROI of a single pixel and filter it - that will be a filter response at that particular pixel (however, it's possible to emulate the old behavior by passing `isolated=false` to `FilterEngine::start` or `FilterEngine::apply`). You can pass the ROI explicitly to `FilterEngine::apply`, or construct a new matrix headers:

```
// compute dI/dx derivative at src(x,y)

// method 1:
// form a matrix header for a single value
float val1 = 0;
Mat dst1(1,1,CV_32F,&val1);

Ptr<FilterEngine> Fx = createDerivFilter(CV_32F, CV_32F,
                        1, 0, 3, BORDER_REFLECT_101);
Fx->apply(src, Rect(x,y,1,1), Point(), dst1);

// method 2:
// form a matrix header for a single value
float val2 = 0;
Mat dst2(1,1,CV_32F,&val2);

Mat pix_roi(src, Rect(x,y,1,1));
Sobel(pix_roi, dst2, dst2.type(), 1, 0, 3, 1, 0, BORDER_REFLECT_101);

printf("method1 = %g, method2 = %g\n", val1, val2);
```

Note on the data types. As it was mentioned in cv::BaseFilter description, the specific filters can process data of any type, despite that `Base*Filter::operator()` only takes `uchar` pointers and no information about the actual types. To make it all work, the following rules are used:

- in case of separable filtering `FilterEngine::rowFilter` applied first. It transforms the input image data (of type `srcType`) to the intermediate results stored in the internal buffers (of type `bufType`). Then these intermediate results are processed *as single-channel data* with `FilterEngine::columnFilter` and stored in the output image (of type `dstType`). Thus, the input type for `rowFilter` is `srcType` and the output type is `bufType`; the input type for `columnFilter` is `CV_MAT_DEPTH(bufType)` and the output type is `CV_MAT_DEPTH(dstType)`.

- in case of non-separable filtering `bufType` must be the same as `srcType`. The source data is copied to the temporary buffer if needed and then just passed to `FilterEngine::filter2D`. That is, the input type for `filter2D` is `srcType` (=`bufType`) and the output type is `dstType`.

See also: cv::BaseColumnFilter, cv::BaseFilter, cv::BaseRowFilter, cv::createBoxFilter, cv::createDerivFilter, cv::createGaussianFilter, cv::createLinearFilter, cv::createMorphologyFilter, cv::createSeparableLinearFilter

---

cv::bilateralFilter

Applies bilateral filter to the image

```
void bilateralFilter( const Mat& src, Mat& dst, int d,
           double sigmaColor, double sigmaSpace,
           int borderType=BORDER_DEFAULT );
```

src The source 8-bit or floating-point, 1-channel or 3-channel image

dst The destination image; will have the same size and the same type as `src`

d The diameter of each pixel neighborhood, that is used during filtering. If it is non-positive, it's computed from `sigmaSpace`

sigmaColor Filter sigma in the color space. Larger value of the parameter means that farther colors within the pixel neighborhood (see `sigmaSpace`) will be mixed together, resulting in larger areas of semi-equal color

sigmaSpace Filter sigma in the coordinate space. Larger value of the parameter means that farther pixels will influence each other (as long as their colors are close enough; see `sigmaColor`). Then `d>0`, it specifies the neighborhood size regardless of `sigmaSpace`, otherwise `d` is proportional to `sigmaSpace`

The function applies bilateral filtering to the input image, as described in http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

---

cv::blur

Smoothes image using normalized box filter

```
void blur( const Mat& src, Mat& dst,
           Size ksize, Point anchor=Point(-1,-1),
           int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same type as `src`

ksize The smoothing kernel size

anchor The anchor point. The default value `Point(-1,-1)` means that the anchor is at the kernel center

borderType The border mode used to extrapolate pixels outside of the image

The function smoothes the image using the kernel:

$$K = \frac{1}{\texttt{ksize.width*ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

The call `blur(src, dst, ksize, anchor, borderType)` is equivalent to `boxFilter(src, dst, src.type(), anchor, true, borderType)`.

See also: cv::boxFilter, cv::bilateralFilter, cv::GaussianBlur, cv::medianBlur.

---

cv::borderInterpolate

Computes source location of extrapolated pixel

```
int borderInterpolate( int p, int len, int borderType );
```

p 0-based coordinate of the extrapolated pixel along one of the axes, likely ¡0 or ¿=`len`

len length of the array along the corresponding axis

borderType the border type, one of the `BORDER_*`, except for `BORDER_TRANSPARENT` and `BORDER_ISOLATED`. When `borderType==BORDER_CONSTANT` the function always returns -1, regardless of `p` and `len`

The function computes and returns the coordinate of the donor pixel, corresponding to the specified extrapolated pixel when using the specified extrapolation border mode. For example, if we use `BORDER_WRAP` mode in the horizontal direction, `BORDER_REFLECT_101` in the vertical direction and want to compute value of the "virtual" pixel `Point(-5, 100)` in a floating-point image `img`, it will be

```
float val = img.at<float>(borderInterpolate(100, img.rows, BORDER_REFLECT_101),
                          borderInterpolate(-5, img.cols, BORDER_WRAP));
```

Normally, the function is not called directly; it is used inside cv::FilterEngine and cv::copyMakeBorder to compute tables for quick extrapolation.

See also: cv::FilterEngine, cv::copyMakeBorder

---

cv::boxFilter

Smoothes image using box filter

```
void boxFilter( const Mat& src, Mat& dst, int ddepth,
            Size ksize, Point anchor=Point(-1,-1),
            bool normalize=true,
            int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same type as `src`

ksize The smoothing kernel size

anchor The anchor point. The default value `Point(-1,-1)` means that the anchor is at the kernel center

normalize Indicates, whether the kernel is normalized by its area or not

borderType The border mode used to extrapolate pixels outside of the image

The function smoothes the image using the kernel:

$$K = \alpha \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

where

$$\alpha = \begin{cases} \frac{1}{\texttt{ksize.width*ksize.height}} & \text{when } \texttt{normalize=true} \\ 1 & \text{otherwise} \end{cases}$$

Unnormalized box filter is useful for computing various integral characteristics over each pixel neighborhood, such as covariation matrices of image derivatives (used in dense optical flow algorithms, Harris corner detector etc.). If you need to compute pixel sums over variable-size windows, use cv::integral.

See also: cv::boxFilter, cv::bilateralFilter, cv::GaussianBlur, cv::medianBlur, cv::integral.

---

cv::buildPyramid

Constructs Gaussian pyramid for an image

```
void buildPyramid( const Mat& src, vector<Mat>& dst, int maxlevel );
```

src The source image; check cv::pyrDown for the list of supported types

dst The destination vector of `maxlevel+1` images of the same type as `src`; `dst[0]` will be the same as `src`, `dst[1]` is the next pyramid layer, a smoothed and down-sized `src` etc.

maxlevel The 0-based index of the last (i.e. the smallest) pyramid layer; it must be non-negative

The function constructs a vector of images and builds the gaussian pyramid by recursively applying cv::pyrDown to the previously built pyramid layers, starting from `dst[0]==src`.

---

cv::copyMakeBorder

Forms a border around the image

```
void copyMakeBorder( const Mat& src, Mat& dst,
            int top, int bottom, int left, int right,
            int borderType, const Scalar& value=Scalar() );
```

src The source image

dst The destination image; will have the same type as `src` and the size `Size(src.cols+left+right, src.rows+top+bottom)`

top, bottom, left, right Specify how much pixels in each direction from the source image rectangle one needs to extrapolate, e.g. `top=1, bottom=1, left=1, right=1` mean that 1 pixel-wide border needs to be built

borderType The border type; see cv::borderInterpolate

value The border value if `borderType==BORDER_CONSTANT`

The function copies the source image into the middle of the destination image. The areas to the left, to the right, above and below the copied source image will be filled with extrapolated pixels. This is not what cv::FilterEngine or based on it filtering functions do (they extrapolate pixels on-fly), but what other more complex functions, including your own, may do to simplify image boundary handling.

The function supports the mode when `src` is already in the middle of `dst`. In this case the function does not copy `src` itself, but simply constructs the border, e.g.:

```
// let border be the same in all directions
int border=2;
// constructs a larger image to fit both the image and the border
Mat gray_buf(rgb.rows + border*2, rgb.cols + border*2, rgb.depth());
// select the middle part of it w/o copying data
Mat gray(gray_canvas, Rect(border, border, rgb.cols, rgb.rows));
// convert image from RGB to grayscale
cvtColor(rgb, gray, CV_RGB2GRAY);
// form a border in-place
copyMakeBorder(gray, gray_buf, border, border,
               border, border, BORDER_REPLICATE);
// now do some custom filtering ...
...
```

See also: cv::borderInterpolate

---

cv::createBoxFilter

Returns box filter engine

```
Ptr<FilterEngine> createBoxFilter( int srcType, int dstType,
           Size ksize, Point anchor=Point(-1,-1),
           bool normalize=true,
```

```
            int borderType=BORDER_DEFAULT);
Ptr<BaseRowFilter> getRowSumFilter(int srcType, int sumType,
            int ksize, int anchor=-1);
Ptr<BaseColumnFilter> getColumnSumFilter(int sumType, int dstType,
            int ksize, int anchor=-1, double scale=1);
```

srcType The source image type

sumType The intermediate horizontal sum type; must have as many channels as `srcType`

dstType The destination image type; must have as many channels as `srcType`

ksize The aperture size

anchor The anchor position with the kernel; negative values mean that the anchor is at the kernel center

normalize Whether the sums are normalized or not; see cv::boxFilter

scale Another way to specify normalization in lower-level `getColumnSumFilter`

borderType Which border type to use; see cv::borderInterpolate

The function is a convenience function that retrieves horizontal sum primitive filter with cv::getRowSumFilter, vertical sum filter with cv::getColumnSumFilter, constructs new cv::FilterEngine and passes both of the primitive filters there. The constructed filter engine can be used for image filtering with normalized or unnormalized box filter.

The function itself is used by cv::blur and cv::boxFilter.

See also: cv::FilterEngine, cv::blur, cv::boxFilter.

---

cv::createDerivFilter

Returns engine for computing image derivatives

```
Ptr<FilterEngine> createDerivFilter( int srcType, int dstType,
            int dx, int dy, int ksize,
            int borderType=BORDER_DEFAULT );
```

srcType The source image type

dstType The destination image type; must have as many channels as `srcType`

dx The derivative order in respect with x

dy The derivative order in respect with y

ksize The aperture size; see cv::getDerivKernels

borderType Which border type to use; see cv::borderInterpolate

The function cv::createDerivFilter is a small convenience function that retrieves linear filter coefficients for computing image derivatives using cv::getDerivKernels and then creates a separable linear filter with cv::createSeparableLinearFilter. The function is used by cv::Sobel and cv::Scharr.

See also: cv::createSeparableLinearFilter, cv::getDerivKernels, cv::Scharr, cv::Sobel.

---

cv::createGaussianFilter

Returns engine for smoothing images with a Gaussian filter

```
Ptr<FilterEngine> createGaussianFilter( int type, Size ksize,
            double sigmaX, double sigmaY=0,
            int borderType=BORDER_DEFAULT);
```

type The source and the destination image type

ksize The aperture size; see cv::getGaussianKernel

sigmaX The Gaussian sigma in the horizontal direction; see cv::getGaussianKernel

sigmaY The Gaussian sigma in the vertical direction; if 0, then `sigmaY` ← `sigmaX`

borderType Which border type to use; see cv::borderInterpolate

The function cv::createGaussianFilter computes Gaussian kernel coefficients and then returns separable linear filter for that kernel. The function is used by cv::GaussianBlur. Note that while the function takes just one data type, both for input and output, you can pass by this limitation by calling cv::getGaussianKernel and then cv::createSeparableFilter directly.

See also: cv::createSeparableLinearFilter, cv::getGaussianKernel, cv::GaussianBlur.

cv::createLinearFilter

Creates non-separable linear filter engine

```
Ptr<FilterEngine> createLinearFilter(int srcType, int dstType,
           const Mat& kernel, Point _anchor=Point(-1,-1),
           double delta=0, int rowBorderType=BORDER_DEFAULT,
           int columnBorderType=-1, const Scalar& borderValue=Scalar());
Ptr<BaseFilter> getLinearFilter(int srcType, int dstType,
           const Mat& kernel,
           Point anchor=Point(-1,-1),
           double delta=0, int bits=0);
```

srcType The source image type

dstType The destination image type; must have as many channels as `srcType`

kernel The 2D array of filter coefficients

anchor The anchor point within the kernel; special value `Point(-1,-1)` means that the anchor is at the kernel center

delta The value added to the filtered results before storing them

bits When the kernel is an integer matrix representing fixed-point filter coefficients, the parameter specifies the number of the fractional bits

rowBorderType, columnBorderType The pixel extrapolation methods in the horizontal and the vertical directions; see cv::borderInterpolate

borderValue Used in case of constant border

The function returns pointer to 2D linear filter for the specified kernel, the source array type and the destination array type. The function is a higher-level function that calls `getLinearFilter` and passes the retrieved 2D filter to cv::FilterEngine constructor.

See also: cv::createSeparableLinearFilter, cv::FilterEngine, cv::filter2D

cv::createMorphologyFilter

Creates engine for non-separable morphological operations

```
Ptr<FilterEngine> createMorphologyFilter(int op, int type,
           const Mat& element, Point anchor=Point(-1,-1),
           int rowBorderType=BORDER_CONSTANT,
           int columnBorderType=-1,
           const Scalar& borderValue=morphologyDefaultBorderValue());
Ptr<BaseFilter> getMorphologyFilter(int op, int type, const Mat&
element,
           Point anchor=Point(-1,-1));
Ptr<BaseRowFilter> getMorphologyRowFilter(int op, int type,
           int esize, int anchor=-1);
Ptr<BaseColumnFilter> getMorphologyColumnFilter(int op, int type,
           int esize, int anchor=-1);
static inline Scalar morphologyDefaultBorderValue()
           return Scalar::all(DBL_MAX);
```

op The morphology operation id, `MORPH_ERODE` or `MORPH_DILATE`

type The input/output image type

element The 2D 8-bit structuring element for the morphological operation. Non-zero elements indicate the pixels that belong to the element

esize The horizontal or vertical structuring element size for separable morphological operations

anchor The anchor position within the structuring element; negative values mean that the anchor is at the center

rowBorderType, columnBorderType The pixel extrapolation methods in the horizontal and the vertical directions; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value, `morphologyDefaultBorderValue`, has the special meaning. It is transformed $+\inf$ for the erosion and to $-\inf$ for the dilation, which means that the minimum (maximum) is effectively computed only over the pixels that are inside the image.

The functions construct primitive morphological filtering operations or a filter engine based on them. Normally it's enough to use cv::createMorphologyFilter or even higher-level cv::erode, cv::dilate or cv::morphologyEx, Note, that cv::createMorphologyFilter analyses the structuring element shape and builds a separable morphological filter engine when the structuring element is square.

See also: cv::erode, cv::dilate, cv::morphologyEx, cv::FilterEngine

---

cv::createSeparableLinearFilter

Creates engine for separable linear filter

```
Ptr<FilterEngine> createSeparableLinearFilter(int srcType, int dstType,
           const Mat& rowKernel, const Mat& columnKernel,
           Point anchor=Point(-1,-1), double delta=0,
           int rowBorderType=BORDER_DEFAULT,
           int columnBorderType=-1,
           const Scalar& borderValue=Scalar());
Ptr<BaseColumnFilter> getLinearColumnFilter(int bufType, int dstType,
           const Mat& columnKernel, int anchor,
           int symmetryType, double delta=0,
           int bits=0);
Ptr<BaseRowFilter> getLinearRowFilter(int srcType, int bufType,
           const Mat& rowKernel, int anchor,
           int symmetryType);
```

srcType The source array type

dstType The destination image type; must have as many channels as `srcType`

bufType The inermediate buffer type; must have as many channels as `srcType`

rowKernel The coefficients for filtering each row

columnKernel The coefficients for filtering each column

anchor The anchor position within the kernel; negative values mean that anchor is positioned at the aperture center

delta The value added to the filtered results before storing them

bits When the kernel is an integer matrix representing fixed-point filter coefficients, the parameter specifies the number of the fractional bits

rowBorderType, columnBorderType The pixel extrapolation methods in the horizontal and the vertical directions; see cv::borderInterpolate

borderValue Used in case of a constant border

symmetryType The type of each of the row and column kernel; see cv::getKernelType.

The functions construct primitive separable linear filtering operations or a filter engine based on them. Normally it's enough to use cv::createSeparableLinearFilter or even higher-level cv::sepFilter2D. The function cv::createMorphologyFilter is smart enough to figure out the `symmetryType` for each of the two kernels, the intermediate `bufType`, and, if the filtering can be done in integer arithmetics, the number of `bits` to encode the filter coefficients. If it does not work for you, it's possible to call `getLinearColumnFilter`, `getLinearRowFilter` directly and then pass them to cv::FilterEngine constructor.

See also: cv::sepFilter2D, cv::createLinearFilter, cv::FilterEngine, cv::getKernelType

---

cv::dilate

Dilates an image by using a specific structuring element.

```
void dilate( const Mat& src, Mat& dst, const Mat& element,
             Point anchor=Point(-1,-1), int iterations=1,
             int borderType=BORDER_CONSTANT,
             const Scalar& borderValue=morphologyDefaultBorderValue() );
```

src The source image

dst The destination image. It will have the same size and the same type as `src`

element The structuring element used for dilation. If `element=Mat()`, a 3×3 rectangular structuring element is used

anchor Position of the anchor within the element. The default value $(-1, -1)$ means that the anchor is at the element center

iterations The number of times dilation is applied

borderType The pixel extrapolation method; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value has a special meaning, see cv::createMorphologyFilter

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\texttt{dst}(x,y) = \max_{(x',y'):\, \texttt{element}(x',y') \neq 0} \texttt{src}(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (`iterations`) times. In the case of multi-channel images each channel is processed independently.

See also: cv::erode, cv::morphologyEx, cv::createMorphologyFilter

---

cv::erode

Erodes an image by using a specific structuring element.

```
void erode( const Mat& src, Mat& dst, const Mat& element,
            Point anchor=Point(-1,-1), int iterations=1,
            int borderType=BORDER_CONSTANT,
            const Scalar& borderValue=morphologyDefaultBorderValue() );
```

src The source image

dst The destination image. It will have the same size and the same type as `src`

element The structuring element used for dilation. If `element=Mat()`, a 3×3 rectangular structuring element is used

anchor Position of the anchor within the element. The default value $(-1, -1)$ means that the anchor is at the element center

iterations The number of times erosion is applied

borderType The pixel extrapolation method; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value has a special meaning, see cv::createMorphoogyFilter

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\texttt{dst}(x,y) = \min_{(x',y'):\, \texttt{element}(x',y') \neq 0} \texttt{src}(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (`iterations`) times. In the case of multi-channel images each channel is processed independently.

See also: cv::dilate, cv::morphologyEx, cv::createMorphologyFilter

---

cv::filter2D

Convolves an image with the kernel

```
void filter2D( const Mat& src, Mat& dst, int ddepth,
           const Mat& kernel, Point anchor=Point(-1,-1),
           double delta=0, int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image. It will have the same size and the same number of channels as `src`

ddepth The desired depth of the destination image. If it is negative, it will be the same as `src.depth()`

kernel Convolution kernel (or rather a correlation kernel), a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cv::split and process them individually

anchor The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor should lie within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center

delta The optional value added to the filtered pixels before storing them in `dst`

borderType The pixel extrapolation method; see cv::borderInterpolate

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

The function does actually computes correlation, not the convolution:

$$\texttt{dst}(x,y) = \sum_{\substack{0\le x'<\texttt{kernel.cols},\\ 0\le y'<\texttt{kernel.rows}}} \texttt{kernel}(x',y') * \texttt{src}(x+x'-\texttt{anchor.x}, y+y'-\texttt{anchor.y})$$

That is, the kernel is not mirrored around the anchor point. If you need a real convolution, flip the kernel using cv::flip and set the new anchor to `(kernel.cols - anchor.x - 1, kernel.rows - anchor.y - 1)`.

The function uses DFT-based algorithm in case of sufficiently large kernels (11×11) and the direct algorithm (that uses the engine retrieved by cv::createLinearFilter) for small kernels.

See also: cv::sepFilter2D, cv::createLinearFilter, cv::dft, cv::matchTemplate

---

cv::GaussianBlur

Smoothes image using a Gaussian filter

```
void GaussianBlur( const Mat& src, Mat& dst, Size ksize,
           double sigmaX, double sigmaY=0,
           int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same type as `src`

ksize The Gaussian kernel size; `ksize.width` and `ksize.height` can differ, but they both must be positive and odd. Or, they can be zero's, then they are computed from `sigma*`

sigmaX, sigmaY The Gaussian kernel standard deviations in X and Y direction. If `sigmaY` is zero, it is set to be equal to `sigmaX`. If they are both zeros, they are computed from `ksize.width` and `ksize.height`, respectively, see cv::getGaussianKernel. To fully control the result regardless of possible future modification of all this semantics, it is recommended to specify all of `ksize`, `sigmaX` and `sigmaY`

borderType The pixel extrapolation method; see cv::borderInterpolate

The function convolves the source image with the specified Gaussian kernel. In-place filtering is supported.

See also: cv::sepFilter2D, cv::filter2D, cv::blur, cv::boxFilter, cv::bilateralFilter, cv::medianBlur

cv::getDerivKernels

Returns filter coefficients for computing spatial image derivatives

```
void getDerivKernels( Mat& kx, Mat& ky, int dx, int dy, int ksize,
           bool normalize=false, int ktype=CV_32F );
```

kx The output matrix of row filter coefficients; will have type `ktype`

ky The output matrix of column filter coefficients; will have type `ktype`

dx The derivative order in respect with x

dy The derivative order in respect with y

ksize The aperture size. It can be `CV_SCHARR`, 1, 3, 5 or 7

normalize Indicates, whether to normalize (scale down) the filter coefficients or not. In theory the coefficients should have the denominator $= 2^{ksize*2-dx-dy-2}$. If you are going to filter floating-point images, you will likely want to use the normalized kernels. But if you compute derivatives of a 8-bit image, store the results in 16-bit image and wish to preserve all the fractional bits, you may want to set `normalize=false`.

ktype The type of filter coefficients. It can be `CV_32f` or `CV_64F`

The function computes and returns the filter coefficients for spatial image derivatives. When `ksize=CV_SCHARR`, the Scharr 3×3 kernels are generated, see cv::Scharr. Otherwise, Sobel kernels are generated, see cv::Sobel. The filters are normally passed to cv::sepFilter2D or to cv::createSeparableLinearFilter.

cv::getGaussianKernel

Returns Gaussian filter coefficients

```
Mat getGaussianKernel( int ksize, double sigma, int ktype=CV_64F );
```

ksize The aperture size. It should be odd ($\texttt{ksize} \mod 2 = 1$) and positive.

sigma The Gaussian standard deviation. If it is non-positive, it is computed from `ksize` as
`sigma = 0.3*(ksize/2 - 1) + 0.8`

ktype The type of filter coefficients. It can be `CV_32f` or `CV_64F`

The function computes and returns the `ksize` $\times 1$ matrix of Gaussian filter coefficients:

$$G_i = \alpha * e^{-(i-(\texttt{ksize}-1)/2)^2/(2*\texttt{sigma})^2},$$

where $i = 0..\texttt{ksize} - 1$ and α is the scale factor chosen so that $\sum_i G_i = 1$
Two of such generated kernels can be passed to cv::sepFilter2D or to cv::createSeparableLinearFilter that will automatically detect that these are smoothing kernels and handle them accordingly. Also you may use the higher-level cv::GaussianBlur.
See also: cv::sepFilter2D, cv::createSeparableLinearFilter, cv::getDerivKernels, cv::getStructuringElement, cv::GaussianBlur.

---

cv::getKernelType

Returns the kernel type

```
int getKernelType(const Mat& kernel, Point anchor);
```

kernel 1D array of the kernel coefficients to analyze

anchor The anchor position within the kernel

The function analyzes the kernel coefficients and returns the corresponding kernel type:

KERNEL_GENERAL Generic kernel - when there is no any type of symmetry or other properties

KERNEL_SYMMETRICAL The kernel is symmetrical: $\texttt{kernel}_i == \texttt{kernel}_{ksize-i-1}$ and the anchor is at the center

KERNEL_ASYMMETRICAL The kernel is asymmetrical: $\texttt{kernel}_i == -\texttt{kernel}_{ksize-i-1}$ and the anchor is at the center

KERNEL_SMOOTH All the kernel elements are non-negative and sum to 1. E.g. the Gaussian kernel is both smooth kernel and symmetrical, so the function will return `KERNEL_SMOOTH | KERNEL_SYMMETRICAL`

KERNEL_INTEGER Al the kernel coefficients are integer numbers. This flag can be combined with `KERNEL_SYMMETRICAL` or `KERNEL_ASYMMETRICAL`

cv::getStructuringElement

Returns the structuring element of the specified size and shape for morphological operations

```
Mat getStructuringElement(int shape, Size esize,
            Point anchor=Point(-1,-1));
```

shape The element shape, one of:

- `MORPH_RECT` - rectangular structuring element

$$E_{ij} = 1$$

- `MORPH_ELLIPSE` - elliptic structuring element, i.e. a filled ellipse inscribed into the rectangle `Rect(0, 0, esize.width, 0.esize.height)`
- `MORPH_CROSS` - cross-shaped structuring element:

$$E_{ij} = \begin{cases} 1 & \text{if i=}\texttt{anchor.y}\text{ or j=}\texttt{anchor.x} \\ 0 & \text{otherwise} \end{cases}$$

esize Size of the structuring element

anchor The anchor position within the element. The default value $(-1, -1)$ means that the anchor is at the center. Note that only the cross-shaped element's shape depends on the anchor position; in other cases the anchor just regulates by how much the result of the morphological operation is shifted

The function constructs and returns the structuring element that can be then passed to cv::createMorphologyF cv::erode, cv::dilate or cv::morphologyEx. But also you can construct an arbitrary binary mask yourself and use it as the structuring element.

cv::medianBlur

Smoothes image using median filter

```
void medianBlur( const Mat& src, Mat& dst, int ksize );
```

src The source 1-, 3- or 4-channel image. When `ksize` is 3 or 5, the image depth should be `CV_8U`, `CV_16U` or `CV_32F`. For larger aperture sizes it can only be `CV_8U`

dst The destination array; will have the same size and the same type as `src`

ksize The aperture linear size. It must be odd and more than 1, i.e. 3, 5, 7 ...

The function smoothes image using the median filter with `ksize` × `ksize` aperture. Each channel of a multi-channel image is processed independently. In-place operation is supported.

See also: cv::bilateralFilter, cv::blur, cv::boxFilter, cv::GaussianBlur

---

cv::morphologyEx

Performs advanced morphological transformations

```
void morphologyEx( const Mat& src, Mat& dst,
           int op, const Mat& element,
           Point anchor=Point(-1,-1), int iterations=1,
           int borderType=BORDER_CONSTANT,
           const Scalar& borderValue=morphologyDefaultBorderValue() );
```

src Source image

dst Destination image. It will have the same size and the same type as `src`

element Structuring element

op Type of morphological operation, one of the following:

- **MORPH_OPEN** opening
- **MORPH_CLOSE** closing
- **MORPH_GRADIENT** morphological gradient
- **MORPH_TOPHAT** "top hat"
- **MORPH_BLACKHAT** "black hat"

iterations Number of times erosion and dilation are applied

borderType The pixel extrapolation method; see cv::borderInterpolate

borderValue The border value in case of a constant border. The default value has a special meaning, see cv::createMorphoogyFilter

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$\texttt{dst} = \mathrm{open}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\mathrm{erode}(\texttt{src}, \texttt{element}))$$

Closing:

$$\texttt{dst} = \mathrm{close}(\texttt{src}, \texttt{element}) = \mathrm{erode}(\mathrm{dilate}(\texttt{src}, \texttt{element}))$$

Morphological gradient:

$$\texttt{dst} = \mathrm{morph\_grad}(\texttt{src}, \texttt{element}) = \mathrm{dilate}(\texttt{src}, \texttt{element}) - \mathrm{erode}(\texttt{src}, \texttt{element})$$

"Top hat":

$$\texttt{dst} = \mathrm{tophat}(\texttt{src}, \texttt{element}) = \texttt{src} - \mathrm{open}(\texttt{src}, \texttt{element})$$

"Black hat":

$$\texttt{dst} = \mathrm{blackhat}(\texttt{src}, \texttt{element}) = \mathrm{close}(\texttt{src}, \texttt{element}) - \texttt{src}$$

Any of the operations can be done in-place.

See also: cv::dilate, cv::erode, cv::createMorphologyFilter

---

cv::Laplacian

Calculates the Laplacian of an image

```
void Laplacian( const Mat& src, Mat& dst, int ddepth,
           int ksize=1, double scale=1, double delta=0,
           int borderType=BORDER_DEFAULT );
```

src Source image

dst Destination image; will have the same size and the same number of channels as `src`

ddepth The desired depth of the destination image

ksize The aperture size used to compute the second-derivative filters, see cv::getDerivKernels. It must be positive and odd

scale The optional scale factor for the computed Laplacian values (by default, no scaling is applied, see cv::getDerivKernels)

delta The optional delta value, added to the results prior to storing them in `dst`

borderType The pixel extrapolation method, see cv::borderInterpolate

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst} = \Delta \texttt{src} = \frac{\partial^2 \texttt{src}}{\partial x^2} + \frac{\partial^2 \texttt{src}}{\partial y^2}$$

This is done when `ksize > 1`. When `ksize == 1`, the Laplacian is computed by filtering the image with the following 3×3 aperture:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

See also: cv::Sobel, cv::Scharr

---

cv::pyrDown

Smoothes an image and downsamples it.

```
void pyrDown( const Mat& src, Mat& dst, const Size& dstsize=Size());
```

src The source image

dst The destination image. It will have the specified size and the same type as `src`

dstsize Size of the destination image. By default it is computed as `Size((src.cols+1)/2, (src.rows+1)/2)`. But in any case the following conditions should be satisfied:

$$|\texttt{dstsize.width} * 2 - src.cols| \leq 2$$
$$|\texttt{dstsize.height} * 2 - src.rows| \leq 2$$

The function performs the downsampling step of the Gaussian pyramid construction. First it convolves the source image with the kernel:

$$\frac{1}{16}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

and then downsamples the image by rejecting even rows and columns.

---

cv::pyrUp

Upsamples an image and then smoothes it

```
void pyrUp( const Mat& src, Mat& dst, const Size& dstsize=Size());
```

src The source image

dst The destination image. It will have the specified size and the same type as `src`

dstsize Size of the destination image. By default it is computed as `Size(src.cols*2, (src.rows*2)`. But in any case the following conditions should be satisfied:

$$|\texttt{dstsize.width} - src.cols * 2| \leq (\texttt{dstsize.width} \mod 2)$$
$$|\texttt{dstsize.height} - src.rows * 2| \leq (\texttt{dstsize.height} \mod 2)$$

The function performs the upsampling step of the Gaussian pyramid construction (it can actually be used to construct the Laplacian pyramid). First it upsamples the source image by injecting even zero rows and columns and then convolves the result with the same kernel as in cv::pyrDown, multiplied by 4.

---

cv::sepFilter2D

Applies separable linear filter to an image

```
void sepFilter2D( const Mat& src, Mat& dst, int ddepth,
           const Mat& rowKernel, const Mat& columnKernel,
           Point anchor=Point(-1,-1),
           double delta=0, int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same number of channels as `src`

ddepth The destination image depth

rowKernel The coefficients for filtering each row

columnKernel The coefficients for filtering each column

anchor The anchor position within the kernel; The default value $(-1, 1)$ means that the anchor is at the kernel center

delta The value added to the filtered results before storing them

borderType The pixel extrapolation method; see cv::borderInterpolate

The function applies a separable linear filter to the image. That is, first, every row of `src` is filtered with 1D kernel `rowKernel`. Then, every column of the result is filtered with 1D kernel `columnKernel` and the final result shifted by `delta` is stored in `dst`.

See also: cv::createSeparableLinearFilter, cv::filter2D, cv::Sobel, cv::GaussianBlur, cv::boxFilter, cv::blur.

---

cv::Sobel

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator

```
void Sobel( const Mat& src, Mat& dst, int ddepth,
            int xorder, int yorder, int ksize=3,
            double scale=1, double delta=0,
            int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same number of channels as `src`

ddepth The destination image depth

xorder Order of the derivative x

yorder Order of the derivative y

ksize Size of the extended Sobel kernel, must be 1, 3, 5 or 7

scale The optional scale factor for the computed derivative values (by default, no scaling is applied, see cv::getDerivKernels)

delta The optional delta value, added to the results prior to storing them in `dst`

borderType The pixel extrapolation method, see cv::borderInterpolate

In all cases except 1, an `ksize` $\times$ `ksize` separable kernel will be used to calculate the derivative. When `ksize = 1`, a 3×1 or 1×3 kernel will be used (i.e. no Gaussian smoothing is done). `ksize = 1` can only be used for the first or the second x- or y- derivatives.

There is also the special value `ksize = CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. The Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\texttt{dst} = \frac{\partial^{xorder+yorder}\texttt{src}}{\partial x^{xorder}\partial y^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation, so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `ksize` = 3) or (`xorder` = 0, `yorder` = 1, `ksize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and the second one corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

See also: cv::Scharr, cv::Lapacian, cv::sepFilter2D, cv::filter2D, cv::GaussianBlur

cv::Scharr

Calculates the first x- or y- image derivative using Scharr operator

```
void Scharr( const Mat& src, Mat& dst, int ddepth,
             int xorder, int yorder,
             double scale=1, double delta=0,
             int borderType=BORDER_DEFAULT );
```

src The source image

dst The destination image; will have the same size and the same number of channels as `src`

ddepth The destination image depth

xorder Order of the derivative x

yorder Order of the derivative y

scale The optional scale factor for the computed derivative values (by default, no scaling is applied, see cv::getDerivKernels)

delta The optional delta value, added to the results prior to storing them in `dst`

borderType The pixel extrapolation method, see cv::borderInterpolate

The function computes the first x- or y- spatial image derivative using Scharr operator. The call

```
Scharr(src, dst, ddepth, xorder, yorder, scale, delta, borderType)
```

is equivalent to

```
Sobel(src, dst, ddepth, xorder, yorder, CV_SCHARR, scale, delta, borderType).
```

8.2 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions

compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x,y) = \texttt{src}(f_x(x,y), f_y(x,y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic cv::remap and to the simplest and the fastest cv::resize, need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x,y) one of $f_x(x,y)$ or $f_y(x,y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method BORDER_TRANSPARENT, which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x,y)$ and $f_y(x,y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x,y), f_y(x,y))$ and then the value of the polynomial at $(f_x(x,y), f_y(x,y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see cv::resize.

---

cv::convertMaps

Converts image transformation maps from one representation to another

```
void convertMaps( const Mat& map1, const Mat& map2,
           Mat& dstmap1, Mat& dstmap2,
           int dstmap1type, bool nninterpolation=false );
```

map1 The first input map of type CV_16SC2 or CV_32FC1 or CV_32FC2

map2 The second input map of type CV_16UC1 or CV_32FC1 or none (empty matrix), respectively

dstmap1 The first output map; will have type `dstmap1type` and the same size as `src`

dstmap2 The second output map

dstmap1type The type of the first output map; should be `CV_16SC2`, `CV_32FC1` or `CV_32FC2`

nninterpolation Indicates whether the fixed-point maps will be used for nearest-neighbor or for more complex interpolation

The function converts a pair of maps for cv::remap from one representation to another. The following options (`(map1.type(), map2.type())` → `(dstmap1.type(), dstmap2.type())`) are supported:

1. `(CV_32FC1, CV_32FC1)` → `(CV_16SC2, CV_16UC1)`. This is the most frequently used conversion operation, in which the original floating-point maps (see cv::remap) are converted to more compact and much faster fixed-point representation. The first output array will contain the rounded coordinates and the second array (created only when `nninterpolation=false`) will contain indices in the interpolation tables.
2. `(CV_32FC2)` → `(CV_16SC2, CV_16UC1)`. The same as above, but the original maps are stored in one 2-channel matrix.
3. the reverse conversion. Obviously, the reconstructed floating-point maps will not be exactly the same as the originals.

See also: cv::remap, cv::undisort, cv::initUndistortRectifyMap

---

cv::getAffineTransform

Calculates the affine transform from 3 pairs of the corresponding points

```
Mat getAffineTransform( const Point2f src[], const Point2f dst[] );
```

src Coordinates of a triangle vertices in the source image

dst Coordinates of the corresponding triangle vertices in the destination image

The function calculates the 2×3 matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

See also: cv::warpAffine, cv::transform

---

cv::getPerspectiveTransform

Calculates the perspective transform from 4 pairs of the corresponding points

```
Mat getPerspectiveTransform( const Point2f src[],
           const Point2f dst[] );
```

src Coordinates of a quadrange vertices in the source image

dst Coordinates of the corresponding quadrangle vertices in the destination image

The function calculates the 3×3 matrix of a perspective transform such that:

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \texttt{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

See also: cv::findHomography, cv::warpPerspective, cv::perspectiveTransform

---

cv::getRectSubPix

Retrieves the pixel rectangle from an image with sub-pixel accuracy

```
void getRectSubPix( const Mat& image, Size patchSize,
           Point2f center, Mat& dst, int patchType=-1 );
```

src Source image

patchSize Size of the extracted patch

center Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image

dst The extracted patch; will have the size `patchSize` and the same number of channels as `src`

patchType The depth of the extracted pixels. By default they will have the same depth as `src`

The function `getRectSubPix` extracts pixels from `src`:

$$dst(x, y) = src(x + \texttt{center.x} - (\texttt{dst.cols} - 1) * 0.5, y + \texttt{center.y} - (\texttt{dst.rows} - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode (see cv::borderInterpolate) is used to extrapolate the pixel values outside of the image.

See also: cv::warpAffine, cv::warpPerspective

---

cv::getRotationMatrix2D

Calculates the affine matrix of 2d rotation.

```
Mat getRotationMatrix2D( Point2f center, double angle, double scale );
```

center Center of the rotation in the source image

angle The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)

scale Isotropic scale factor

The function calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} - (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\begin{aligned} \alpha &= \texttt{scale} \cdot \cos \texttt{angle}, \\ \beta &= \texttt{scale} \cdot \sin \texttt{angle} \end{aligned}$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

See also: cv::getAffineTransform, cv::warpAffine, cv::transform

---

cv::invertAffineTransform

Inverts an affine transformation

```
void invertAffineTransform(const Mat& M, Mat& iM);
```

M The original affine transformation

iM The output reverse affine transformation

The function computes inverse affine transformation represented by 2×3 matrix M:

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

The result will also be a 2×3 matrix of the same type as M.

---

cv::cv::remap

Applies a generic geometrical transformation to an image.

```
void remap( const Mat& src, Mat& dst, const Mat& map1, const Mat& map2,
            int interpolation, int borderMode=BORDER_CONSTANT,
            const Scalar& borderValue=Scalar());
```

src Source image

dst Destination image. It will have the same size as map1 and the same type as src

map1 The first map of either (x,y) points or just x values having type CV_16SC2, CV_32FC1 or CV_32FC2. See cv::convertMaps for converting floating point representation to fixed-point for speed.

map2 The second map of y values having type CV_16UC1, CV_32FC1 or none (empty map if map1 is (x,y) points), respectively

interpolation The interpolation method, see cv::resize. The method INTER_AREA is not supported by this function

borderMode The pixel extrapolation method, see cv::borderInterpolate. When the borderMode=BORDER_TRANSPARENT, it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function

borderValue A value used in the case of a constant border. By default it is 0

The function remap transforms the source image using the specified map:

$$\texttt{dst}(x,y) = \texttt{src}(map_x(x,y), map_y(x,y))$$

Where values of pixels with non-integer coordinates are computed using one of the available interpolation methods. map_x and map_y can be encoded as separate floating-point maps in map_1 and map_2 respectively, or interleaved floating-point maps of (x,y) in map_1, or fixed-point maps made by using cv::convertMaps. The reason you might want to convert from floating to fixed-point representations of a map is that they can yield much faster (2x) remapping operations. In the converted case, map_1 contains pairs (cvFloor(x), cvFloor(y)) and map_2 contains indices in a table of interpolation coefficients.

This function can not operate in-place.

---

cv::cv::resize

Resizes an image

```
void resize( const Mat& src, Mat& dst,
             Size dsize, double fx=0, double fy=0,
             int interpolation=INTER_LINEAR );
```

src Source image

dst Destination image. It will have size dsize (when it is non-zero) or the size computed from src.size() and fx and fy. The type of dst will be the same as of src.

dsize The destination image size. If it is zero, then it is computed as:

```
dsize = Size(round(fx*src.cols), round(fy*src.rows))
```

. Either dsize or both fx or fy must be non-zero.

fx The scale factor along the horizontal axis. When 0, it is computed as

```
(double)dsize.width/src.cols
```

fy The scale factor along the vertical axis. When 0, it is computed as

```
(double)dsize.height/src.rows
```

interpolation The interpolation method:

- **INTER_NEAREST** nearest-neighbor interpolation
- **INTER_LINEAR** bilinear interpolation (used by default)
- **INTER_AREA** resampling using pixel area relation. It may be the preferred method for image decimation, as it gives moire-free results. But when the image is zoomed, it is similar to the INTER_NEAREST method
- **INTER_CUBIC** bicubic interpolation over 4x4 pixel neighborhood
- **INTER_LANCZOS4** Lanczos interpolation over 8x8 pixel neighborhood

The function resize resizes an image src down to or up to the specified size. Note that the initial dst type or size are not taken into account. Instead the size and type are derived from the src, dsize, fx and fy. If you want to resize src so that it fits the pre-created dst, you may call the function as:

```
// explicitly specify dsize=dst.size(); fx and fy will be computed from that.
resize(src, dst, dst.size(), 0, 0, interpolation);
```

If you want to decimate the image by factor of 2 in each direction, you can call the function this way:

```
// specify fx and fy and let the function to compute the destination image size.
resize(src, dst, Size(), 0.5, 0.5, interpolation);
```

See also: cv::warpAffine, cv::warpPerspective, cv::remap.

---

cv::warpAffine

Applies an affine transformation to an image.

```
void warpAffine( const Mat& src, Mat& dst,
            const Mat& M, Size dsize,
            int flags=INTER_LINEAR,
            int borderMode=BORDER_CONSTANT,
            const Scalar& borderValue=Scalar());
```

src Source image

dst Destination image; will have size dsize and the same type as src

M 2×3 transformation matrix

dsize Size of the destination image

flags A combination of interpolation methods, see cv::resize, and the optional flag WARP_INVERSE_MAP that means that M is the inverse transformation (dst $\rightarrow$ src)

borderMode The pixel extrapolation method, see cv::borderInterpolate. When the borderMode=BORDER_TRANSPARENT, it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function

borderValue A value used in case of a constant border. By default it is 0

The function warpAffine transforms the source image using the specified matrix:

$$\mathtt{dst}(x,y) = \mathtt{src}(\mathtt{M}_{11}x + \mathtt{M}_{12}y + \mathtt{M}_{13}, \mathtt{M}_{21}x + \mathtt{M}_{22}y + \mathtt{M}_{23})$$

when the flag WARP_INVERSE_MAP is set. Otherwise, the transformation is first inverted with cv::invertAffineTransform and then put in the formula above instead of M. The function can not operate in-place.

See also: cv::warpPerspective, cv::resize, cv::remap, cv::getRectSubPix, cv::transform

---

cv::warpPerspective

Applies a perspective transformation to an image.

```
void warpPerspective( const Mat& src, Mat& dst,
           const Mat& M, Size dsize,
           int flags=INTER_LINEAR,
           int borderMode=BORDER_CONSTANT,
           const Scalar& borderValue=Scalar());
```

src Source image

dst Destination image; will have size dsize and the same type as src

M 3×3 transformation matrix

dsize Size of the destination image

flags A combination of interpolation methods, see cv::resize, and the optional flag WARP_INVERSE_MAP that means that M is the inverse transformation (dst → src)

borderMode The pixel extrapolation method, see cv::borderInterpolate. When the borderMode=BORDER_TRANSPARENT, it means that the pixels in the destination image that corresponds to the "outliers" in the source image are not modified by the function

borderValue A value used in case of a constant border. By default it is 0

The function warpPerspective transforms the source image using the specified matrix:

$$\texttt{dst}(x,y) = \texttt{src}\left(\frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}}\right)$$

when the flag WARP_INVERSE_MAP is set. Otherwise, the transformation is first inverted with cv::invert and then put in the formula above instead of M. The function can not operate in-place.

See also: cv::warpAffine, cv::resize, cv::remap, cv::getRectSubPix, cv::perspectiveTransform

8.3 Miscellaneous Image Transformations

---

cv::adaptiveThreshold

Applies an adaptive threshold to an array.

```
void adaptiveThreshold( const Mat& src, Mat& dst, double maxValue,
            int adaptiveMethod, int thresholdType,
            int blockSize, double C );
```

src Source 8-bit single-channel image

dst Destination image; will have the same size and the same type as src

maxValue The non-zero value assigned to the pixels for which the condition is satisfied. See the discussion

adaptiveMethod Adaptive thresholding algorithm to use, ADAPTIVE_THRESH_MEAN_C or ADAPTIVE_THRESH_G (see the discussion)

thresholdType Thresholding type; must be one of THRESH_BINARY or THRESH_BINARY_INV

blockSize The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

C The constant subtracted from the mean or weighted mean (see the discussion); normally, it's positive, but may be zero or negative as well

The function transforms a grayscale image to a binary image according to the formulas:

THRESH_BINARY

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

1. For the method ADAPTIVE_THRESH_MEAN_C the threshold value $T(x,y)$ is the mean of a blockSize $\times$ blockSize neighborhood of (x,y), minus C.
2. For the method ADAPTIVE_THRESH_GAUSSIAN_C the threshold value $T(x,y)$ is the weighted sum (i.e. cross-correlation with a Gaussian window) of a blockSize $\times$ blockSize neighborhood of (x,y), minus C. The default sigma (standard deviation) is used for the specified blockSize, see cv::getGaussianKernel.

The function can process the image in-place.
See also: cv::threshold, cv::blur, cv::GaussianBlur

---

cv::cvtColor

Converts image from one color space to another

```
void cvtColor( const Mat& src, Mat& dst, int code, int dstCn=0 );
```

src The source image, 8-bit unsigned, 16-bit unsigned (CV_16UC...) or single-precision floating-point

dst The destination image; will have the same size and the same depth as src

code The color space conversion code; see the discussion

dstCn The number of channels in the destination image; if the parameter is 0, the number of the channels will be derived automatically from `src` and the `code`

The function converts the input image from one color space to another. In the case of transformation to-from RGB color space the ordering of the channels should be specified explicitly (RGB or BGR).

The conventional ranges for R, G and B channel values are:

- 0 to 255 for `CV_8U` images
- 0 to 65535 for `CV_16U` images and
- 0 to 1 for `CV_32F` images.

Of course, in the case of linear transformations the range does not matter, but in the non-linear cases the input RGB image should be normalized to the proper value range in order to get the correct results, e.g. for RGB→L*u*v* transformation. For example, if you have a 32-bit floating-point image directly converted from 8-bit image without any scaling, then it will have 0..255 value range, instead of the assumed by the function 0..1. So, before calling `cvtColor`, you need first to scale the image down:

```
img *= 1./255;
cvtColor(img, img, CV_BGR2Luv);
```

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} \quad Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} \quad R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

  ```
  cvtColor(src, bwsrc, CV_RGB2GRAY);
  ```

  Some more advanced channel reordering can also be done with cv::mixChannels.

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB`):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB $\leftrightarrow$ YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB`)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

  where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

  Y, Cr and Cb cover the whole value range.

- RGB $\leftrightarrow$ HSV (`CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V = R \\ 120 + 60(B - R)/S & \text{if } V = G \\ 240 + 60(R - G)/S & \text{if } V = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$

On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R, G, B)$$

$$V_{min} \leftarrow min(R, G, B)$$

$$L \leftarrow \frac{V_{max} - V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max} - V_{min}}{V_{max} + V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max} - V_{min}}{2 - (V_{max} + V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V_{max} = R \\ 120 + 60(B - R)/S & \text{if } V_{max} = G \\ 240 + 60(R - G)/S & \text{if } V_{max} = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$ On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255 \cdot V, S \leftarrow 255 \cdot S, H \leftarrow H/2 \text{ (to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535 \cdot V, S < -65535 \cdot S, H < -H$$

32-bit images H, S, V are left as is

- RGB $\leftrightarrow$ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \le 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t \le 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \le L \le 100$, $-127 \le a \le 127$, $-127 \le b \le 127$

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow L * 255/100, \ a \leftarrow a + 128, \ b \leftarrow b + 128$$

16-bit images currently not supported

32-bit images L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y \le 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow 255/100L, \; u \leftarrow 255/354(u + 134), \; v \leftarrow 255/256(v + 140)$$

16-bit images currently not supported

32-bit images L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Charles Poynton site http://www.poynton.com/ColorFAQ.html

- Bayer → RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerGB2RGB, CV_BayerRG2RGB, CV_BayerGR2RGB) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{matrix} R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants CV_Bayer C_1C_2 2BGR and CV_Bayer C_1C_2 2RGB indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

cv::distanceTransform

Calculates the distance to the closest zero pixel for each pixel of the source image.

```
void distanceTransform( const Mat& src, Mat& dst,
           int distanceType, int maskSize );
void distanceTransform( const Mat& src, Mat& dst, Mat& labels,
           int distanceType, int maskSize );
```

src 8-bit, single-channel (binary) source image

dst Output image with calculated distances; will be 32-bit floating-point, single-channel image of the same size as `src`

distanceType Type of distance; can be `CV_DIST_L1, CV_DIST_L2` or `CV_DIST_C`

maskSize Size of the distance transform mask; can be 3, 5 or `CV_DIST_MASK_PRECISE` (the latter option is only supported by the first of the functions). In the case of `CV_DIST_L1` or `CV_DIST_C` distance type the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 or any larger aperture.

labels The optional output 2d array of labels - the discrete Voronoi diagram; will have type `CV_32SC1` and the same size as `src`. See the discussion

The functions `distanceTransform` calculate the approximate or precise distance from every binary image pixel to the nearest zero pixel. (for zero image pixels the distance will obviously be zero).

When `maskSize == CV_DIST_MASK_PRECISE` and `distanceType == CV_DIST_L2`, the function runs the algorithm described in [**?**].

In other cases the algorithm [?] is used, that is, for pixel the function finds the shortest path to the nearest zero pixel consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (that is denoted as `a`), all the diagonal shifts must have the same cost (denoted `b`), and all knight's moves must have the same cost (denoted `c`). For `CV_DIST_C` and `CV_DIST_L1` types the distance is calculated precisely, whereas for `CV_DIST_L2` (Euclidian distance) the distance can be calculated only with some relative error (a 5×5 mask gives more accurate results). For `a`, `b` and `c` OpenCV uses the values suggested in the original paper:

CV_DIST_C	(3×3)	a = 1, b = 1
CV_DIST_L1	(3×3)	a = 1, b = 2
CV_DIST_L2	(3×3)	a=0.955, b=1.3693
CV_DIST_L2	(5×5)	a=1, b=1.4, c=2.1969

Typically, for a fast, coarse distance estimation CV_DIST_L2, a 3×3 mask is used, and for a more accurate distance estimation CV_DIST_L2, a 5×5 mask or the precise algorithm is used. Note that both the precise and the approximate algorithms are linear on the number of pixels.

The second variant of the function does not only compute the minimum distance for each pixel (x, y), but it also identifies the nearest the nearest connected component consisting of zero pixels. Index of the component is stored in labels(x, y). The connected components of zero pixels are also found and marked by the function.

In this mode the complexity is still linear. That is, the function provides a very fast way to compute Voronoi diagram for the binary image. Currently, this second variant can only use the approximate distance transform algorithm.

---

cv::floodFill

Fills a connected component with the given color.

```
int floodFill( Mat& image,
           Point seed, Scalar newVal, Rect* rect=0,
           Scalar loDiff=Scalar(), Scalar upDiff=Scalar(),
           int flags=4 );
int floodFill( Mat& image, Mat& mask,
           Point seed, Scalar newVal, Rect* rect=0,
           Scalar loDiff=Scalar(), Scalar upDiff=Scalar(),
           int flags=4 );
```

image Input/output 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the FLOODFILL_MASK_ONLY flag is set (in the second variant of the function; see below)

mask (For the second function only) Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller. The function uses and updates the mask, so the user takes responsibility of initializing the mask content. Flood-filling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. It is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap. **Note**: because the mask is larger than the filled image, a pixel (x, y) in image will correspond to the pixel $(x + 1, y + 1)$ in the mask

seed The starting point

newVal New value of the repainted domain pixels

loDiff Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component

upDiff Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component

rect The optional output parameter that the function sets to the minimum bounding rectangle of the repainted domain

flags The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:

FLOODFILL_FIXED_RANGE if set, the difference between the current pixel and seed pixel is considered, otherwise the difference between neighbor pixels is considered (i.e. the range is floating)

FLOODFILL_MASK_ONLY (for the second variant only) if set, the function does not change the image (`newVal` is ignored), but fills the mask

The functions `floodFill` fill a connected component starting from the seed point with the specified color. The connectivity is determined by the color/brightness closeness of the neighbor pixels. The pixel at (x, y) is considered to belong to the repainted domain if:

grayscale image, floating range

$$\texttt{src}(x',y') - \texttt{loDiff} \leq \texttt{src}(x,y) \leq \texttt{src}(x',y') + \texttt{upDiff}$$

grayscale image, fixed range

$$\texttt{src}(\texttt{seed}.x,\texttt{seed}.y) - \texttt{loDiff} \leq \texttt{src}(x,y) \leq \texttt{src}(\texttt{seed}.x,\texttt{seed}.y) + \texttt{upDiff}$$

color image, floating range

$$\texttt{src}(x',y')_r - \texttt{loDiff}_r \leq \texttt{src}(x,y)_r \leq \texttt{src}(x',y')_r + \texttt{upDiff}_r$$

$$\texttt{src}(x',y')_g - \texttt{loDiff}_g \leq \texttt{src}(x,y)_g \leq \texttt{src}(x',y')_g + \texttt{upDiff}_g$$

$$\texttt{src}(x',y')_b - \texttt{loDiff}_b \leq \texttt{src}(x,y)_b \leq \texttt{src}(x',y')_b + \texttt{upDiff}_b$$

color image, fixed range

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_r - \texttt{loDiff}_r \leq \texttt{src}(x,y)_r \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_r + \texttt{upDiff}_r$$

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_g - \texttt{loDiff}_g \leq \texttt{src}(x,y)_g \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_g + \texttt{upDiff}_g$$

$$\texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_b - \texttt{loDiff}_b \leq \texttt{src}(x,y)_b \leq \texttt{src}(\texttt{seed}.x, \texttt{seed}.y)_b + \texttt{upDiff}_b$$

where $src(x', y')$ is the value of one of pixel neighbors that is already known to belong to the component. That is, to be added to the connected component, a pixel's color/brightness should be close enough to the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

By using these functions you can either mark a connected component with the specified color in-place, or build a mask and then extract the contour or copy the region to another image etc. Various modes of the function are demonstrated in `floodfill.c` sample.

See also: cv::findContours

---

cv::inpaint

Inpaints the selected region in the image.

```
void inpaint( const Mat& src, const Mat& inpaintMask,
           Mat& dst, double inpaintRadius, int flags );
```

src The input 8-bit 1-channel or 3-channel image.

inpaintMask The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.

dst The output image; will have the same size and the same type as `src`

inpaintRadius The radius of a circlular neighborhood of each point inpainted that is considered by the algorithm.

flags The inpainting method, one of the following:

INPAINT_NS Navier-Stokes based method.

INPAINT_TELEA The method by Alexandru Telea [?]

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video. See http://en.wikipedia.org/wiki/Inpainting for more details.

---

cv::integral

Calculates the integral of an image.

```
void integral( const Mat& image, Mat& sum, int sdepth=-1 );
void integral( const Mat& image, Mat& sum, Mat& sqsum, int sdepth=-1 );
void integral( const Mat& image, Mat& sum,
           Mat& sqsum, Mat& tilted, int sdepth=-1 );
```

image The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

sum The integral image, $(W+1) \times (H+1)$, 32-bit integer or floating-point (32f or 64f)

sqsum The integral image for squared pixel values, $(W+1) \times (H+1)$, double precision floating-point (64f)

tilted The integral for the image rotated by 45 degrees, $(W+1) \times (H+1)$, the same data type as `sum`

sdepth The desired depth of the integral and the tilted integral images, `CV_32S`, `CV_32F` or `CV_64F`

The functions `integral` calculate one or more integral images for the source image as following:

$$\texttt{sum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)$$

$$\texttt{sqsum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)^2$$

$$\texttt{tilted}(X,Y) = \sum_{y<Y,abs(x-X)<y} \texttt{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum_{x_1 \leq x < x_2,\, y_1 \leq y < y_2} \texttt{image}(x,y) = \texttt{sum}(x_2,y_2) - \texttt{sum}(x_1,y_2) - \texttt{sum}(x_2,y_1) + \texttt{sum}(x_1,x_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

---

cv::threshold

Applies a fixed-level threshold to each array element

```
double threshold( const Mat& src, Mat& dst, double thresh,
           double maxVal, int thresholdType );
```

src Source array (single-channel, 8-bit of 32-bit floating point)

dst Destination array; will have the same size and the same type as `src`

thresh Threshold value

maxVal Maximum value to use with `THRESH_BINARY` and `THRESH_BINARY_INV` thresholding types

thresholdType Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (cv::compare could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by `thresholdType`:

THRESH_BINARY

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxVal} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{maxVal} & \text{otherwise} \end{cases}$$

THRESH_TRUNC

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

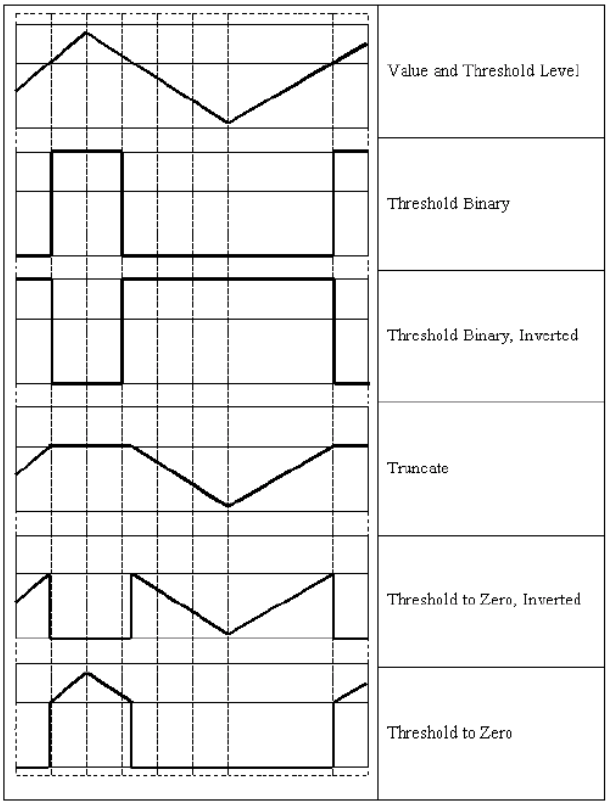
THRESH_TOZERO

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ 0 & \text{otherwise} \end{cases}$$

THRESH_TOZERO_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{thresh} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Also, the special value THRESH_OTSU may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified thresh. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.



See also: cv::adaptiveThreshold, cv::findContours, cv::compare, cv::min, cv::max

---

cv::watershed

Does marker-based image segmentation using watershed algrorithm

```
void watershed( const Mat& image, Mat& markers );
```

image The input 8-bit 3-channel image.

markers The input/output 32-bit single-channel image (map) of markers. It should have the same size as `image`

The function implements one of the variants of watershed, non-parametric marker-based segmentation algorithm, described in [**?**]. Before passing the image to the function, user has to outline roughly the desired regions in the image `markers` with positive (> 0) indices, i.e. every region

is represented as one or more connected components with the pixel values 1, 2, 3 etc (such markers can be retrieved from a binary mask using cv::findContoursand cv::drawContours, see `watershed.cpp` demo). The markers will be "seeds" of the future image regions. All the other pixels in `markers`, which relation to the outlined regions is not known and should be defined by the algorithm, should be set to 0's. On the output of the function, each pixel in markers is set to one of values of the "seed" components, or to -1 at boundaries between the regions.

Note, that it is not necessary that every two neighbor connected components are separated by a watershed boundary (-1's pixels), for example, in case when such tangent components exist in the initial marker image. Visual demonstration and usage example of the function can be found in OpenCV samples directory; see `watershed.cpp` demo.

See also: cv::findContours

8.4 Histograms

---

cv::calcHist

Calculates histogram of a set of arrays

```
void calcHist( const Mat* arrays, int narrays,
           const int* channels, const Mat& mask,
           MatND& hist, int dims, const int* histSize,
           const float** ranges, bool uniform=true,
           bool accumulate=false );
void calcHist( const Mat* arrays, int narrays,
           const int* channels, const Mat& mask,
           SparseMat& hist, int dims, const int* histSize,
           const float** ranges, bool uniform=true,
           bool accumulate=false );
```

arrays Source arrays. They all should have the same depth, `CV_8U` or `CV_32F`, and the same size. Each of them can have an arbitrary number of channels

narrays The number of source arrays

channels The list of `dims` channels that are used to compute the histogram. The first array channels are numerated from 0 to `arrays[0].channels()-1`, the second array channels are counted from `arrays[0].channels()` to `arrays[0].channels() + arrays[1].channels()-1` etc.

mask The optional mask. If the matrix is not empty, it must be 8-bit array of the same size as `arrays[i]`. The non-zero mask elements mark the array elements that are counted in the histogram

hist The output histogram, a dense or sparse `dims`-dimensional array

dims The histogram dimensionality; must be positive and not greater than `CV_MAX_DIMS`(=32 in the current OpenCV version)

histSize The array of histogram sizes in each dimension

ranges The array of `dims` arrays of the histogram bin boundaries in each dimension. When the histogram is uniform (`uniform`=true), then for each dimension `i` it's enough to specify the lower (inclusive) boundary L_0 of the 0-th histogram bin and the upper (exclusive) boundary $U_{\texttt{histSize}[i]-1}$ for the last histogram bin `histSize[i]-1`. That is, in the case of uniform histogram each of `ranges[i]` is an array of 2 elements. When the histogram is not uniform (`uniform=false`), then each of `ranges[i]` contains `histSize[i]+1` elements: $L_0, U_0 = L_1, U_1 = L_2, ..., U_{\texttt{histSize[i]}-2} = L_{\texttt{histSize[i]}-1}, U_{\texttt{histSize[i]}-1}$. The array elements, which are not between L_0 and $U_{\texttt{histSize[i]}-1}$, are not counted in the histogram

uniform Indicates whether the histogram is uniform or not, see above

accumulate Accumulation flag. If it is set, the histogram is not cleared in the beginning (when it is allocated). This feature allows user to compute a single histogram from several sets of arrays, or to update the histogram in time

The functions `calcHist` calculate the histogram of one or more arrays. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input arrays. The sample below shows how to compute 2D Hue-Saturation histogram for a color imag

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat src;
    if( argc != 2 || !(src=imread(argv[1], 1)).data )
        return -1;

    Mat hsv;
    cvtColor(src, hsv, CV_BGR2HSV);
```

```
    // let's quantize the hue to 30 levels
    // and the saturation to 32 levels
    int hbins = 30, sbins = 32;
    int histSize[] = {hbins, sbins};
    // hue varies from 0 to 179, see cvtColor
    float hranges[] = { 0, 180 };
    // saturation varies from 0 (black-gray-white) to
    // 255 (pure spectrum color)
    float sranges[] = { 0, 256 };
    float* ranges[] = { hranges, sranges };
    MatND hist;
    // we compute the histogram from the 0-th and 1-st channels
    int channels[] = {0, 1};

    calcHist( &hsv, 1, channels, Mat(), // do not use mask
        hist, 2, histSize, ranges,
        true, // the histogram is uniform
        false );
    double maxVal=0;
    minMaxLoc(hist, 0, &maxVal, 0, 0);

    int scale = 10;
    Mat histImg = Mat::zeros(sbins*scale, hbins*10, CV_8UC3);

    for( int h = 0; h < hbins; h++ )
        for( int s = 0; s < sbins; s++ )
        {
            float binVal = hist.at<float>(h, s);
            int intensity = cvRound(binVal*255/maxValue);
            cvRectangle( histImg, Point(h*scale, s*scale),
                        Point( (h+1)*scale - 1, (s+1)*scale - 1),
                        Scalar::all(intensity),
                        CV_FILLED );
        }

    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "H-S Histogram", 1 );
    imshow( "H-S Histogram", histImg );

    waitKey();
}
```

cv::calcBackProject

Calculates the back projection of a histogram.

```
void calcBackProject( const Mat* arrays, int narrays,
           const int* channels, const MatND& hist,
           Mat& backProject, const float** ranges,
           double scale=1, bool uniform=true );
void calcBackProject( const Mat* arrays, int narrays,
           const int* channels, const SparseMat& hist,
           Mat& backProject, const float** ranges,
           double scale=1, bool uniform=true );
```

arrays Source arrays. They all should have the same depth, `CV_8U` or `CV_32F`, and the same size. Each of them can have an arbitrary number of channels

narrays The number of source arrays

channels The list of channels that are used to compute the back projection. The number of channels must match the histogram dimensionality. The first array channels are numerated from 0 to `arrays[0].channels()-1`, the second array channels are counted from `arrays[0].channels()` to `arrays[0].channels() + arrays[1].channels()-1` etc.

hist The input histogram, a dense or sparse

backProject Destination back projection aray; will be a single-channel array of the same size and the same depth as `arrays[0]`

ranges The array of arrays of the histogram bin boundaries in each dimension. See cv::calcHist

scale The optional scale factor for the output back projection

uniform Indicates whether the histogram is uniform or not, see above

The functions `calcBackProject` calculate the back project of the histogram. That is, similarly to `calcHist`, at each location `(x, y)` the function collects the values from the selected channels in the input images and finds the corresponding histogram bin. But instead of incrementing it, the function reads the bin value, scales it by `scale` and stores in `backProject(x,y)`. In terms of statistics, the function computes probability of each element value in respect with the empirical probability distribution represented by the histogram. Here is how, for example, you can find and track a bright-colored object in a scene:

1. Before the tracking, show the object to the camera such that covers almost the whole frame. Calculate a hue histogram. The histogram will likely have a strong maximums, corresponding to the dominant colors in the object.

2. During the tracking, calculate back projection of a hue plane of each input video frame using that pre-computed histogram. Threshold the back projection to suppress weak colors. It may also have sense to suppress pixels with non sufficient color saturation and too dark or too bright pixels.

3. Find connected components in the resulting picture and choose, for example, the largest component.

That is the approximate algorithm of cv::CAMShift color object tracker.
See also: cv::calcHist

---

cv::compareHist

Compares two histograms

```
double compareHist( const MatND& H1, const MatND& H2, int method );
double compareHist( const SparseMat& H1,
           const SparseMat& H2, int method );
```

H1 The first compared histogram

H2 The second compared histogram of the same size as `H1`

method The comparison method, one of the following:

CV_COMP_CORREL Correlation

CV_COMP_CHISQR Chi-Square

CV_COMP_INTERSECT Intersection

CV_COMP_BHATTACHARYYA Bhattacharyya distance

The functions `compareHist` compare two dense or two sparse histograms using the specified method:

Correlation (method=CV_COMP_CORREL)

$$d(H_1, H_2) = \frac{\sum_I (H_1(I) - \bar{H_1})(H_2(I) - \bar{H_2})}{\sqrt{\sum_I (H_1(I) - \bar{H_1})^2 \sum_I (H_2(I) - \bar{H_2})^2}}$$

where

$$\bar{H_k} = \frac{1}{N} \sum_J H_k(J)$$

and N is the total number of histogram bins.

Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \frac{1}{\sqrt{\bar{H_1} \bar{H_2} N^2}} \sum_I \sqrt{H_1(I) \cdot H_2(I)}}$$

The function returns $d(H_1, H_2)$.

While the function works well with 1-, 2-, 3-dimensional dense histograms, it may not be suitable for high-dimensional sparse histograms, where, because of aliasing and sampling problems the coordinates of non-zero histogram bins can slightly shift. To compare such histograms or more general sparse configurations of weighted points, consider using the cv::calcEMD function.

---

cv::equalizeHist

Equalizes the histogram of a grayscale image.

```
void equalizeHist( const Mat& src, Mat& dst );
```

src The source 8-bit single channel image

dst The destination image; will have the same size and the same type as `src`

The function equalizes the histogram of the input image using the following algorithm:

1. calculate the histogram H for `src`.
2. normalize the histogram so that the sum of histogram bins is 255.
3. compute the integral of the histogram:
$$H'_i = \sum_{0 \le j < i} H(j)$$
4. transform the image using H' as a look-up table: $\texttt{dst}(x,y) = H'(\texttt{src}(x,y))$

The algorithm normalizes the brightness and increases the contrast of the image.

8.5 Feature Detection

---

cv::Canny

Finds edges in an image using Canny algorithm.

```
void Canny( const Mat& image, Mat& edges,
            double threshold1, double threshold2,
            int apertureSize=3, bool L2gradient=false );
```

image Single-channel 8-bit input image

edges The output edge map. It will have the same size and the same type as `image`

threshold1 The first threshold for the hysteresis procedure

threshold2 The second threshold for the hysteresis procedure

apertureSize Aperture size for the cv::Sobel operator

L2gradient Indicates, whether the more accurate L_2 norm $= \sqrt{(dI/dx)^2 + (dI/dy)^2}$ should be used to compute the image gradient magnitude (`L2gradient=true`), or a faster default L_1 norm $= |dI/dx| + |dI/dy|$ is enough (`L2gradient=false`)

The function finds edges in the input image `image` and marks them in the output map `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges, see http://en.wikipedia.org/wiki/Canny_edge_detector

cv::cornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
void cornerEigenValsAndVecs( const Mat& src, Mat& dst,
           int blockSize, int apertureSize,
           int borderType=BORDER_DEFAULT );
```

src Input single-channel 8-bit or floating-point image

dst Image to store the results. It will have the same size as `src` and the type `CV_32FC(6)`

blockSize Neighborhood size (see discussion)

apertureSize Aperture parameter for the cv::Sobel operator

boderType Pixel extrapolation method; see cv::borderInterpolate

For every pixel p, the function `cornerEigenValsAndVecs` considers a `blockSize` × `blockSize` neigborhood $S(p)$. It calculates the covariation matrix of derivatives over the neighborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dxdI/dy)^2 \\ \sum_{S(p)}(dI/dxdI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

Where the derivatives are computed using cv::Sobel operator.

After that it finds eigenvectors and eigenvalues of M and stores them into destination image in the form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the eigenvalues of M; not sorted

x_1, y_1 are the eigenvectors corresponding to λ_1

x_2, y_2 are the eigenvectors corresponding to λ_2

The output of the function can be used for robust edge or corner detection.

See also: cv::cornerMinEigenVal, cv::cornerHarris, cv::preCornerDetect

cv::cornerHarris

Harris edge detector.

```
void cornerHarris( const Mat& src, Mat& dst, int blockSize,
           int apertureSize, double k,
           int borderType=BORDER_DEFAULT );
```

src Input single-channel 8-bit or floating-point image

dst Image to store the Harris detector responses; will have type `CV_32FC1` and the same size as `src`

blockSize Neighborhood size (see the discussion of cv::cornerEigenValsAndVecs)

apertureSize Aperture parameter for the cv::Sobel operator

k Harris detector free parameter. See the formula below

boderType Pixel extrapolation method; see cv::borderInterpolate

The function runs the Harris edge detector on the image. Similarly to cv::cornerMinEigenVal and cv::cornerEigenValsAndVecs, for each pixel (x, y) it calculates a 2×2 gradient covariation matrix $M^{(x,y)}$ over a `blockSize` $\times$ `blockSize` neighborhood. Then, it computes the following characteristic:

$$\texttt{dst}(x,y) = \det M^{(x,y)} - k \cdot \left(\operatorname{tr} M^{(x,y)} \right)^2$$

Corners in the image can be found as the local maxima of this response map.

---

cv::cornerMinEigenVal

Calculates the minimal eigenvalue of gradient matrices for corner detection.

```
void cornerMinEigenVal( const Mat& src, Mat& dst,
           int blockSize, int apertureSize=3,
           int borderType=BORDER_DEFAULT );
```

src Input single-channel 8-bit or floating-point image

dst Image to store the minimal eigenvalues; will have type `CV_32FC1` and the same size as `src`

blockSize Neighborhood size (see the discussion of cv::cornerEigenValsAndVecs)

apertureSize Aperture parameter for the cv::Sobel operator

boderType Pixel extrapolation method; see cv::borderInterpolate

The function is similar to cv::cornerEigenValsAndVecs but it calculates and stores only the minimal eigenvalue of the covariation matrix of derivatives, i.e. $\min(\lambda_1, \lambda_2)$ in terms of the formulae in cv::cornerEigenValsAndVecs description.

---

cv::cornerSubPix

Refines the corner locations.

```
void cornerSubPix( const Mat& image, vector<Point2f>& corners,
           Size winSize, Size zeroZone,
           TermCriteria criteria );
```

image Input image

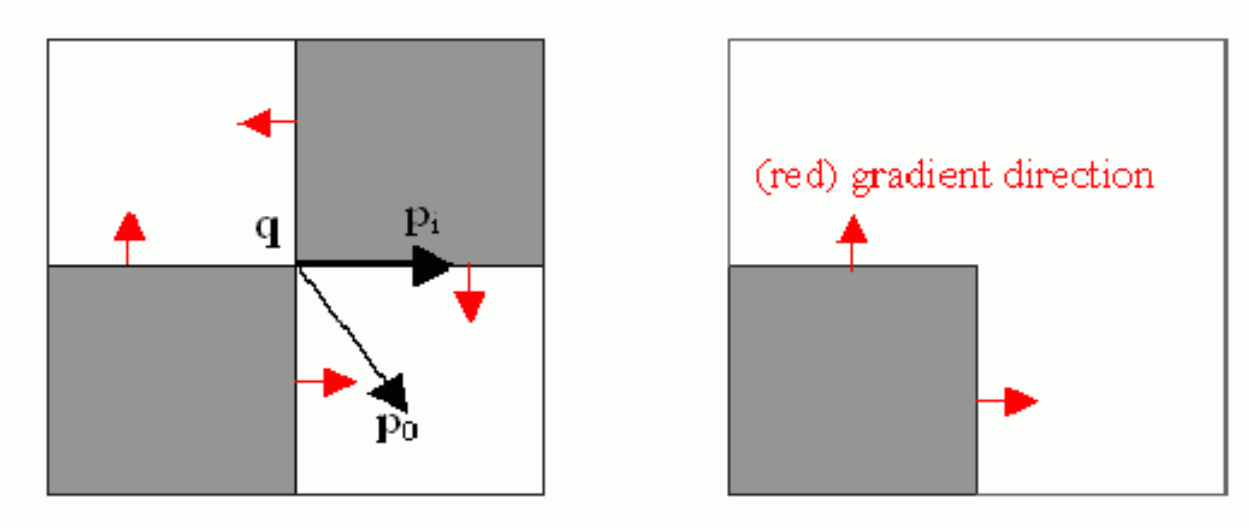
corners Initial coordinates of the input corners; refined coordinates on output

winSize Half of the side length of the search window. For example, if `winSize=Size(5,5)`, then a $5*2+1 \times 5*2+1 = 11 \times 11$ search window would be used

zeroZone Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size

criteria Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below.



Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider the expression:

$$\epsilon_i = DI_{p_i}{}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot DI_{p_i}{}^T) - \sum_i (DI_{p_i} \cdot DI_{p_i}{}^T \cdot p_i)$$

where the gradients are summed within a neighborhood ("search window") of q. Calling the first gradient term G and the second gradient term b gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center q and then iterates until the center keeps within a set threshold.

---

cv::goodFeaturesToTrack

Determines strong corners on an image.

```
void goodFeaturesToTrack( const Mat& image, vector<Point2f>& corners,
           int maxCorners, double qualityLevel, double minDistance,
           const Mat& mask=Mat(), int blockSize=3,
           bool useHarrisDetector=false, double k=0.04 );
```

image The input 8-bit or floating-point 32-bit, single-channel image

corners The output vector of detected corners

maxCorners The maximum number of corners to return. If there are more corners than that will be found, the strongest of them will be returned

qualityLevel Characterizes the minimal accepted quality of image corners; the value of the parameter is multiplied by the by the best corner quality measure (which is the min eigenvalue, see cv::cornerMinEigenVal, or the Harris function response, see cv::cornerHarris). The corners, which quality measure is less than the product, will be rejected. For example, if the best corner has the quality measure = 1500, and the `qualityLevel=0.01`, then all the corners which quality measure is less than 15 will be rejected.

minDistance The minimum possible Euclidean distance between the returned corners

mask The optional region of interest. If the image is not empty (then it needs to have the type `CV_8UC1` and the same size as `image`), it will specify the region in which the corners are detected

blockSize Size of the averaging block for computing derivative covariation matrix over each pixel neighborhood, see cv::cornerEigenValsAndVecs

useHarrisDetector Indicates, whether to use Harris operator or cv::cornerMinEigenVal

k Free parameter of Harris detector

The function finds the most prominent corners in the image or in the specified image region, as described in [**?**]:

1. the function first calculates the corner quality measure at every source image pixel using the cv::cornerMinEigenVal or cv::cornerHarris

2. then it performs non-maxima suppression (the local maxima in 3×3 neighborhood are retained).

3. the next step rejects the corners with the minimal eigenvalue less than `qualityLevel` · $\max_{x,y} qualityMeasureMap(x, y)$.

4. the remaining corners are then sorted by the quality measure in the descending order.

5. finally, the function throws away each corner pt_j if there is a stronger corner pt_i ($i < j$) such that the distance between them is less than `minDistance`

The function can be used to initialize a point-based tracker of an object.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel`, and `A` ¿ B, the vector of returned corners with `qualityLevel=A` will be the prefix of the output vector with `qualityLevel=B`.

See also: cv::cornerMinEigenVal, cv::cornerHarris, cv::calcOpticalFlowPyrLK, cv::estimateRigidMotion, cv::PlanarObjectDetector, cv::OneWayDescriptor

---

cv::HoughCircles

Finds circles in a grayscale image using a Hough transform.

```
void HoughCircles( Mat& image, vector<Vec3f>& circles,
           int method, double dp, double minDist,
           double param1=100, double param2=100,
           int minRadius=0, int maxRadius=0 );
```

`image` The 8-bit, single-channel, grayscale input image

`circles` The output vector of found circles. Each vector is encoded as 3-element floating-point vector $(x, y, radius)$

`method` Currently, the only implemented method is `CV_HOUGH_GRADIENT`, which is basically *21HT*, described in [**?**].

`dp` The inverse ratio of the accumulator resolution to the image resolution. For example, if `dp=1`, the accumulator will have the same resolution as the input image, if `dp=2` - accumulator will have half as big width and height, etc

`minDist` Minimum distance between the centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed

param1 The first method-specific parameter. in the case of CV_HOUGH_GRADIENT it is the higher threshold of the two passed to cv::Canny edge detector (the lower one will be twice smaller)

param2 The second method-specific parameter. in the case of CV_HOUGH_GRADIENT it is the accumulator threshold at the center detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first

minRadius Minimum circle radius

maxRadius Maximum circle radius

The function finds circles in a grayscale image using some modification of Hough transform. Here is a short usage example:

```
#include <cv.h>
#include <highgui.h>
#include <math.h>

using namespace cv;

int main(int argc, char** argv)
{
    Mat img, gray;
    if( argc != 2 && !(img=imread(argv[1], 1)).data)
        return -1;
    cvtColor(img, gray, CV_BGR2GRAY);
    // smooth it, otherwise a lot of false circles may be detected
    GaussianBlur( gray, gray, 9, 9, 2, 2 );
    vector<Vec3f> circles;
    houghCircles(gray, circles, CV_HOUGH_GRADIENT,
                 2, gray->rows/4, 200, 100 );
    for( size_t i = 0; i < circles.size(); i++ )
    {
         Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
         int radius = cvRound(circles[i][2]);
         // draw the circle center
         circle( img, center, 3, Scalar(0,255,0), -1, 8, 0 );
         // draw the circle outline
         circle( img, center, radius, Scalar(0,0,255), 3, 8, 0 );
    }
    namedWindow( "circles", 1 );
    imshow( "circles", img );
    return 0;
}
```

Note that usually the function detects the circles' centers well, however it may fail to find the correct radii. You can assist the function by specifying the radius range (`minRadius` and `maxRadius`) if you know it, or you may ignore the returned radius, use only the center and find the correct radius using some additional procedure.

See also: cv::fitEllipse, cv::minEnclosingCircle

---

cv::HoughLines

Finds lines in a binary image using standard Hough transform.

```
void HoughLines( Mat& image, vector<Vec2f>& lines,
            double rho, double theta, int threshold,
            double srn=0, double stn=0 );
```

image The 8-bit, single-channel, binary source image. The image may be modified by the function

lines The output vector of lines. Each line is represented by a two-element vector (ρ, θ). ρ is the distance from the coordinate origin $(0, 0)$ (top-left corner of the image) and θ is the line rotation angle in radians ($0 \sim$ vertical line, $\pi/2 \sim$ horizontal line)

rho Distance resolution of the accumulator in pixels

theta Angle resolution of the accumulator in radians

threshold The accumulator threshold parameter. Only those lines are returned that get enough votes ($>$ `threshold`)

srn For the multi-scale Hough transform it is the divisor for the distance resolution `rho`. The coarse accumulator distance resolution will be `rho` and the accurate accumulator resolution will be `rho/srn`. If both `srn=0` and `stn=0` then the classical Hough transform is used, otherwise both these parameters should be positive.

stn For the multi-scale Hough transform it is the divisor for the distance resolution `theta`

The function implements standard or standard multi-scale Hough transform algorithm for line detection. See cv::HoughLinesP for the code example.

cv::HoughLinesP

Finds lines segments in a binary image using probabilistic Hough transform.

```
void HoughLinesP( Mat& image, vector<Vec4i>& lines,
           double rho, double theta, int threshold,
           double minLineLength=0, double maxLineGap=0 );
```

image The 8-bit, single-channel, binary source image. The image may be modified by the function

lines The output vector of lines. Each line is represented by a 4-element vector (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) are the ending points of each line segment detected.

rho Distance resolution of the accumulator in pixels

theta Angle resolution of the accumulator in radians

threshold The accumulator threshold parameter. Only those lines are returned that get enough votes ($>$ `threshold`)

minLineLength The minimum line length. Line segments shorter than that will be rejected

maxLineGap The maximum allowed gap between points on the same line to link them.

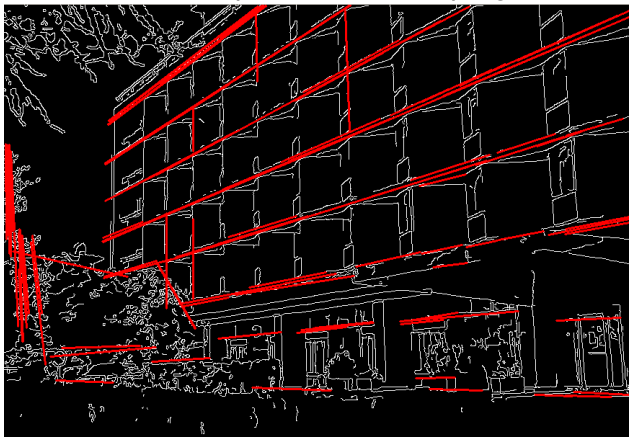
The function implements probabilistic Hough transform algorithm for line detection, described in [**?**]. Below is line detection example:

```
/* This is a standalone program. Pass an image name as a first parameter
of the program.  Switch between standard and probabilistic Hough transform
by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

using namespace cv;

int main(int argc, char** argv)
{
    Mat src, dst, color_dst;
    if( argc != 2 || !(src=imread(argv[1], 0)).data)
        return -1;
```

```
    Canny( src, dst, 50, 200, 3 );
    cvtColor( dst, color_dst, CV_GRAY2BGR );

#if 0
    vector<Vec2f> lines;
    HoughLines( dst, lines, 1, CV_PI/180, 100 );

    for( size_t i = 0; i < lines.size(); i++ )
    {
        float rho = lines[i][0];
        float theta = lines[i][1];
        double a = cos(theta), b = sin(theta);
        double x0 = a*rho, y0 = b*rho;
        Point pt1(cvRound(x0 + 1000*(-b)),
                  cvRound(y0 + 1000*(a)));
        Point pt2(cvRound(x0 - 1000*(-b)),
                  cvRound(y0 - 1000*(a)));
        line( color_dst, pt1, pt2, Scalar(0,0,255), 3, 8 );
    }
#else
    vector<Vec4i> lines;
    HoughLinesP( dst, lines, 1, CV_PI/180, 80, 30, 10 );
    for( size_t i = 0; i < lines.size(); i++ )
    {
        line( color_dst, Point(lines[i][0], lines[i][1]),
            Point(lines[i][2], lines[i][3]), Scalar(0,0,255), 3, 8 );
    }
#endif
    namedWindow( "Source", 1 );
    imshow( "Source", src );

    namedWindow( "Detected Lines", 1 );
    imshow( "Detected Lines", color_dst );

    waitKey(0);
    return 0;
}
```

This is the sample picture the function parameters have been tuned for:

And this is the output of the above program in the case of probabilistic Hough transform



---

cv::perCornerDetect

Calculates the feature map for corner detection

```
void preCornerDetect( const Mat& src, Mat& dst, int apertureSize,
            int borderType=BORDER_DEFAULT );
```

src The source single-channel 8-bit of floating-point image

dst The output image; will have type `CV_32F` and the same size as `src`

apertureSize Aperture size of cv::Sobel

borderType The pixel extrapolation method; see cv::borderInterpolate

The function calculates the complex spatial derivative-based function of the source image

$$\texttt{dst} = (D_x \texttt{src})^2 \cdot D_{yy} \texttt{src} + (D_y \texttt{src})^2 \cdot D_{xx} \texttt{src} - 2 D_x \texttt{src} \cdot D_y \texttt{src} \cdot D_{xy} \texttt{src}$$

where D_x, D_y are the first image derivatives, D_{xx}, D_{yy} are the second image derivatives and D_{xy} is the mixed derivative.

The corners can be found as local maximums of the functions, as shown below:

```
Mat corners, dilated_corners;
preCornerDetect(image, corners, 3);
// dilation with 3x3 rectangular structuring element
dilate(corners, dilated_corners, Mat(), 1);
Mat corner_mask = corners == dilated_corners;
```

KeyPoint

Data structure for salient point detectors

```
KeyPoint
{
public:
    // default constructor
    KeyPoint();
    // two complete constructors
    KeyPoint(Point2f _pt, float _size, float _angle=-1,
            float _response=0, int _octave=0, int _class_id=-1);
    KeyPoint(float x, float y, float _size, float _angle=-1,
             float _response=0, int _octave=0, int _class_id=-1);
    // coordinate of the point
    Point2f pt;
    // feature size
    float size;
    // feature orintation in degrees
    // (has negative value if the orientation
    // is not defined/not computed)
    float angle;
    // feature strength
    // (can be used to select only
    // the most prominent key points)
    float response;
    // scale-space octave in which the feature has been found;
    // may correlate with the size
```

```
    int octave;
    // point (can be used by feature
    // classifiers or object detectors)
    int class_id;
};

// reading/writing a vector of keypoints to a file storage
void write(FileStorage& fs, const string& name, const vector<KeyPoint>& keypoints);
void read(const FileNode& node, vector<KeyPoint>& keypoints);
```

MSER

Maximally-Stable Extremal Region Extractor

```
class MSER : public CvMSERParams
{
public:
    // default constructor
    MSER();
    // constructor that initializes all the algorithm parameters
    MSER( int _delta, int _min_area, int _max_area,
          float _max_variation, float _min_diversity,
          int _max_evolution, double _area_threshold,
          double _min_margin, int _edge_blur_size );
    // runs the extractor on the specified image; returns the MSERs,
    // each encoded as a contour (vector<Point>, see findContours)
    // the optional mask marks the area where MSERs are searched for
    void operator()(Mat& image, vector<vector<Point> >& msers, const Mat& mask) const;
};
```

The class encapsulates all the parameters of MSER (see http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions) extraction algorithm.

SURF

Class for extracting Speeded Up Robust Features from an image.

```
class SURF : public CvSURFParams
{
public:
    // default constructor
    SURF();
    // constructor that initializes all the algorithm parameters
    SURF(double _hessianThreshold, int _nOctaves=4,
```

```
        int _nOctaveLayers=2, bool _extended=false);
    // returns the number of elements in each descriptor (64 or 128)
    int descriptorSize() const;
    // detects keypoints using fast multi-scale Hessian detector
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints) const;
    // detects keypoints and computes the SURF descriptors for them
    void operator()(const Mat& img, const Mat& mask,
                    vector<KeyPoint>& keypoints,
                    vector<float>& descriptors,
                    bool useProvidedKeypoints=false) const;
};
```

The class `SURF` implements Speeded Up Robust Features descriptor [**?**]. There is fast multi-scale Hessian keypoint detector that can be used to find the keypoints (which is the default option), but the descriptors can be also computed for the user-specified keypoints. The function can be used for object tracking and localization, image stitching etc. See the `find_obj.cpp` demo in OpenCV samples directory.

StarDetector

Implements Star keypoint detector

```
class StarDetector : CvStarDetectorParams
{
public:
    // default constructor
    StarDetector();
    // the full constructor initialized all the algorithm parameters:
    // maxSize - maximum size of the features. The following
    //      values of the parameter are supported:
    //      4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
    // responseThreshold - threshold for the approximated laplacian,
    //      used to eliminate weak features. The larger it is,
    //      the less features will be retrieved
    // lineThresholdProjected - another threshold for the laplacian to
    //      eliminate edges
    // lineThresholdBinarized - another threshold for the feature
    //      size to eliminate edges.
    // The larger the 2 threshold, the more points you get.
    StarDetector(int maxSize, int responseThreshold,
                 int lineThresholdProjected,
                 int lineThresholdBinarized,
                 int suppressNonmaxSize);
```

```
    // finds keypoints in an image
    void operator()(const Mat& image, vector<KeyPoint>& keypoints) const;
};
```

The class implements a modified version of CenSurE keypoint detector described in [**?**]

8.6 Motion Analysis and Object Tracking

---

cv::accumulate

Adds image to the accumulator.

```
void accumulate( const Mat& src, Mat& dst, const Mat& mask=Mat() );
```

src The input image, 1- or 3-channel, 8-bit or 32-bit floating point

dst The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds `src`, or some of its elements, to `dst`:

$$\texttt{dst}(x,y) \leftarrow \texttt{dst}(x,y) + \texttt{src}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.

The functions `accumulate*` can be used, for example, to collect statistic of background of a scene, viewed by a still camera, for the further foreground-background segmentation.

See also: cv::accumulateSquare, cv::accumulateProduct, cv::accumulateWeighted

---

cv::accumulateSquare

Adds the square of the source image to the accumulator.

```
void accumulateSquare( const Mat& src, Mat& dst,
           const Mat& mask=Mat() );
```

src The input image, 1- or 3-channel, 8-bit or 32-bit floating point

dst The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the input image `src` or its selected region, raised to power 2, to the accumulator `dst`:

$$\texttt{dst}(x,y) \leftarrow \texttt{dst}(x,y) + \texttt{src}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.
See also: cv::accumulateSquare, cv::accumulateProduct, cv::accumulateWeighted

---

cv::accumulateProduct

Adds the per-element product of two input images to the accumulator.

```
void accumulateProduct( const Mat& src1, const Mat& src2,
           Mat& dst, const Mat& mask=Mat() );
```

src1 The first input image, 1- or 3-channel, 8-bit or 32-bit floating point

src2 The second input image of the same type and the same size as `src1`

dst Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `dst`:

$$\texttt{dst}(x,y) \leftarrow \texttt{dst}(x,y) + \texttt{src1}(x,y) \cdot \texttt{src2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

The function supports multi-channel images; each channel is processed independently.
See also: cv::accumulate, cv::accumulateSquare, cv::accumulateWeighted

---

cv::accumulateWeighted

Updates the running average.

```
void accumulateWeighted( const Mat& src, Mat& dst,
           double alpha, const Mat& mask=Mat() );
```

src The input image, 1- or 3-channel, 8-bit or 32-bit floating point

dst The accumulator image with the same number of channels as input image, 32-bit or 64-bit floating-point

alpha Weight of the input image

mask Optional operation mask

The function calculates the weighted sum of the input image `src` and the accumulator `dst` so that `dst` becomes a running average of frame sequence:

$$\texttt{dst}(x,y) \leftarrow (1 - \texttt{alpha}) \cdot \texttt{dst}(x,y) + \texttt{alpha} \cdot \texttt{src}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

that is, `alpha` regulates the update speed (how fast the accumulator "forgets" about earlier images). The function supports multi-channel images; each channel is processed independently.

See also: cv::accumulate, cv::accumulateSquare, cv::accumulateProduct

---

cv::calcOpticalFlowPyrLK

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids

```
void calcOpticalFlowPyrLK( const Mat& prevImg, const Mat& nextImg,
           const vector<Point2f>& prevPts, vector<Point2f>& nextPts,
           vector<uchar>& status, vector<float>& err,
           Size winSize=Size(15,15), int maxLevel=3,
           TermCriteria criteria=TermCriteria(
           TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01),
           double derivLambda=0.5, int flags=0 );
```

prevImg The first 8-bit single-channel or 3-channel input image

nextImg The second input image of the same size and the same type as `prevImg`

prevPts Vector of points for which the flow needs to be found

nextPts The output vector of points containing the calculated new positions of the input features in the second image

status The output status vector. Each element of the vector is set to 1 if the flow for the corresponding features has been found, 0 otherwise

err The output vector that will contain the difference between patches around the original and moved points

winSize Size of the search window at each pyramid level

maxLevel 0-based maximal pyramid level number. If 0, pyramids are not used (single level), if 1, two levels are used etc.

criteria Specifies the termination criteria of the iterative search algorithm (after the specified maximum number of iterations `criteria.maxCount` or when the search window moves by less than `criteria.epsilon`

derivLambda The relative weight of the spatial image derivatives impact to the optical flow estimation. If `derivLambda=0`, only the image intensity is used, if `derivLambda=1`, only derivatives are used. Any other values between 0 and 1 means that both derivatives and the image intensity are used (in the corresponding proportions).

flags The operation flags:

OPTFLOW_USE_INITIAL_FLOW use initial estimations stored in `nextPts`. If the flag is not set, then initially `nextPts` ← `prevPts`

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids, see [?].

---

cv::calcOpticalFlowFarneback

Computes dense optical flow using Gunnar Farneback's algorithm

```
void calcOpticalFlowFarneback( const Mat& prevImg, const Mat& nextImg,
           Mat& flow, double pyrScale, int levels, int winsize,
           int iterations, int polyN, double polySigma, int flags );
```

prevImg The first 8-bit single-channel input image

nextImg The second input image of the same size and the same type as `prevImg`

flow The computed flow image; will have the same size as `prevImg` and type `CV_32FC2`

pyrScale Specifies the image scale (¡1) to build the pyramids for each image. `pyrScale=0.5` means the classical pyramid, where each next layer is twice smaller than the previous

levels The number of pyramid layers, including the initial image. `levels=1` means that no extra layers are created and only the original images are used

winsize The averaging window size; The larger values increase the algorithm robustness to image noise and give more chances for fast motion detection, but yield more blurred motion field

iterations The number of iterations the algorithm does at each pyramid level

polyN Size of the pixel neighborhood used to find polynomial expansion in each pixel. The larger values mean that the image will be approximated with smoother surfaces, yielding more robust algorithm and more blurred motion field. Typically, `polyN`=5 or 7

polySigma Standard deviation of the Gaussian that is used to smooth derivatives that are used as a basis for the polynomial expansion. For `polyN=5` you can set `polySigma=1.1`, for `polyN=7` a good value would be `polySigma=1.5`

flags The operation flags; can be a combination of the following:

OPTFLOW_USE_INITIAL_FLOW Use the input `flow` as the initial flow approximation

OPTFLOW_FARNEBACK_GAUSSIAN Use a Gaussian `winsize` × `winsize` filter instead of box filter of the same size for optical flow estimation. Usually, this option gives more accurate flow than with a box filter, at the cost of lower speed (and normally `winsize` for a Gaussian window should be set to a larger value to achieve the same level of robustness)

The function finds optical flow for each `prevImg` pixel using the alorithm so that

$$\texttt{prevImg}(x,y) \sim \texttt{nextImg}(\texttt{flow}(x,y)[0], \texttt{flow}(x,y)[1])$$

---

cv::updateMotionHistory

Updates the motion history image by a moving silhouette.

```
void updateMotionHistory( const Mat& silhouette, Mat& mhi,
            double timestamp, double duration );
```

silhouette Silhouette mask that has non-zero pixels where the motion occurs

mhi Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp Current time in milliseconds or other units

duration Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current `timestamp`, while the pixels where motion happened last time a long time ago are cleared.

The function, together with cv::calcMotionGradient and cv::calcGlobalOrientation, implements the motion templates technique, described in [**?**] and [**?**]. See also the OpenCV sample `motempl.c` that demonstrates the use of all the motion template functions.

---

cv::calcMotionGradient

Calculates the gradient orientation of a motion history image.

```
void calcMotionGradient( const Mat& mhi, Mat& mask,
            Mat& orientation,
            double delta1, double delta2,
            int apertureSize=3 );
```

mhi Motion history single-channel floating-point image

mask The output mask image; will have the type `CV_8UC1` and the same size as `mhi`. Its non-zero elements will mark pixels where the motion gradient data is correct

orientation The output motion gradient orientation image; will have the same type and the same size as `mhi`. Each pixel of it will the motion orientation in degrees, from 0 to 360.

delta1, delta2 The minimal and maximal allowed difference between `mhi` values within a pixel neighorhood. That is, the function finds the minimum ($m(x,y)$) and maximum ($M(x,y)$) `mhi` values over 3×3 neighborhood of each pixel and marks the motion orientation at (x,y) as valid only if

$$\min(\texttt{delta1}, \texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1}, \texttt{delta2}).$$

apertureSize The aperture size of cv::Sobel operator

The function calculates the gradient orientation at each pixel (x,y) as:

$$\texttt{orientation}(x,y) = \arctan \frac{d\texttt{mhi}/dy}{d\texttt{mhi}/dx}$$

(in fact, cv::fastArctan and cv::phase are used, so that the computed angle is measured in degrees and covers the full range 0..360). Also, the `mask` is filled to indicate pixels where the computed angle is valid.

---

cv::calcGlobalOrientation

Calculates the global motion orientation in some selected region.

```
double calcGlobalOrientation( const Mat& orientation, const Mat& mask,
           const Mat& mhi, double timestamp,
           double duration );
```

orientation Motion gradient orientation image, calculated by the function cv::calcMotionGradient

mask Mask image. It may be a conjunction of a valid gradient mask, also calculated by cv::calcMotionGradient, and the mask of the region, whose direction needs to be calculated

mhi The motion history image, calculated by cv::updateMotionHistory

timestamp The timestamp passed to cv::updateMotionHistory

duration Maximal duration of motion track in milliseconds, passed to cv::updateMotionHistory

The function calculates the average motion direction in the selected region and returns the angle between 0 degrees and 360 degrees. The average direction is computed from the weighted orientation histogram, where a recent motion has larger weight and the motion occurred in the past has smaller weight, as recorded in `mhi`.

cv::CamShift

Finds the object center, size, and orientation

```
RotatedRect CamShift( const Mat& probImage, Rect& window,
            TermCriteria criteria );
```

probImage Back projection of the object histogram; see cv::calcBackProject

window Initial search window

criteria Stop criteria for the underlying cv::meanShift

The function implements the CAMSHIFT object tracking algrorithm [?]. First, it finds an object center using cv::meanShift and then adjust the window size and finds the optimal rotation. The function returns the rotated rectangle structure that includes the object position, size and the orientation. The next position of the search window can be obtained with `RotatedRect::boundingRect()`.

See the OpenCV sample `camshiftdemo.c` that tracks colored objects.

cv::meanShift

Finds the object on a back projection image.

```
int meanShift( const Mat& probImage, Rect& window,
            TermCriteria criteria );
```

probImage Back projection of the object histogram; see cv::calcBackProject

window Initial search window

criteria The stop criteria for the iterative search algorithm

The function implements iterative object search algorithm. It takes the object back projection on input and the initial position. The mass center in `window` of the back projection image is computed and the search window center shifts to the mass center. The procedure is repeated until the specified number of iterations `criteria.maxCount` is done or until the window center shifts by less than `criteria.epsilon`. The algorithm is used inside cv::CamShift and, unlike cv::CamShift, the search window size or orientation do not change during the search. You can

simply pass the output of cv::calcBackProject to this function, but better results can be obtained if you pre-filter the back projection and remove the noise (e.g. by retrieving connected components with cv::findContours, throwing away contours with small area (cv::contourArea) and rendering the remaining contours with cv::drawContours)

KalmanFilter

Kalman filter class

```
class KalmanFilter
{
public:
    KalmanFilter();newline
    KalmanFilter(int dynamParams, int measureParams, int controlParams=0);newline
    void init(int dynamParams, int measureParams, int controlParams=0);newline
    // predicts statePre from statePost
    const Mat& predict(const Mat& control=Mat());newline
    // corrects statePre based on the input measurement vector
    // and stores the result to statePost.
    const Mat& correct(const Mat& measurement);newline

    Mat statePre;           // predicted state (x'(k)):
                            //    x(k)=A*x(k-1)+B*u(k)
    Mat statePost;          // corrected state (x(k)):
                            //    x(k)=x'(k)+K(k)*(z(k)-H*x'(k))
    Mat transitionMatrix;   // state transition matrix (A)
    Mat controlMatrix;      // control matrix (B)
                            //   (it is not used if there is no control)
    Mat measurementMatrix;  // measurement matrix (H)
    Mat processNoiseCov;    // process noise covariance matrix (Q)
    Mat measurementNoiseCov;// measurement noise covariance matrix (R)
    Mat errorCovPre;        // priori error estimate covariance matrix (P'(k)):
                            //    P'(k)=A*P(k-1)*At + Q)*/
    Mat gain;               // Kalman gain matrix (K(k)):
                            //    K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)
    Mat errorCovPost;       // posteriori error estimate covariance matrix (P(k)):
                            //    P(k)=(I-K(k)*H)*P'(k)
    ...
};
```

The class implements standard Kalman filter http://en.wikipedia.org/wiki/Kalman_filter. However, you can modify `transitionMatrix`, `controlMatrix` and `measurementMatrix` to get the extended Kalman filter functionality. See the OpenCV sample `kalman.c`

8.7 Structural Analysis and Shape Descriptors

cv::moments

Calculates all of the moments up to the third order of a polygon or rasterized shape.

```
Moments moments( const Mat& array, bool binaryImage=false );
```

where the class `Moments` is defined as:

```
class Moments
{
public:
    Moments();
    Moments(double m00, double m10, double m01, double m20, double m11,
            double m02, double m30, double m21, double m12, double m03 );
    Moments( const CvMoments\& moments );
    operator CvMoments() const;

    // spatial moments
    double  m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;
    // central moments
    double  mu20, mu11, mu02, mu30, mu21, mu12, mu03;
    // central normalized moments
    double  nu20, nu11, nu02, nu30, nu21, nu12, nu03;
};
```

array A raster image (single-channel, 8-bit or floating-point 2D array) or an array ($1 \times N$ or $N \times 1$) of 2D points (`Point` or `Point2f`)

binaryImage (For images only) If it is true, then all the non-zero image pixels are treated as 1's

The function computes moments, up to the 3rd order, of a vector shape or a rasterized shape. In case of a raster image, the spatial moments `Moments::`$\mathtt{m}_{ji}$ are computed as:

$$\mathtt{m}_{ji} = \sum_{x,y} \left(\mathtt{array}(x,y) \cdot x^j \cdot y^i \right),$$

the central moments `Moments::`$\mathtt{mu}_{ji}$ are computed as:

$$\mathtt{mu}_{ji} = \sum_{x,y} \left(\mathtt{array}(x,y) \cdot (x - \bar{x})^j \cdot (y - \bar{y})^i \right)$$

where $(\bar{x}, \bar{y})$ is the mass center:

$$\bar{x} = \frac{\mathrm{m}_{10}}{\mathrm{m}_{00}}, \; \bar{y} = \frac{\mathrm{m}_{01}}{\mathrm{m}_{00}}$$

and the normalized central moments $\texttt{Moments::nu}_{ij}$ are computed as:

$$\mathrm{nu}_{ji} = \frac{\mathrm{mu}_{ji}}{\mathrm{m}_{00}^{(i+j)/2+1}}.$$

Note that $\mathrm{mu}_{00} = \mathrm{m}_{00}$, $\mathrm{nu}_{00} = 1$ $\mathrm{nu}_{10} = \mathrm{mu}_{10} = \mathrm{mu}_{01} = \mathrm{mu}_{10} = 0$, hence the values are not stored.

The moments of a contour are defined in the same way, but computed using Green's formula (see http://en.wikipedia.org/wiki/Green_theorem), therefore, because of a limited raster resolution, the moments computed for a contour will be slightly different from the moments computed for the same contour rasterized.

See also: cv::contourArea, cv::arcLength

---

cv::HuMoments

Calculates the seven Hu invariants.

```
void HuMoments( const Moments& moments, double h[7] );
```

moments The input moments, computed with cv::moments

h The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment, that are defined as:

$$\begin{aligned}
h[0] &= \eta_{20} + \eta_{02} \\
h[1] &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
h[2] &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
h[3] &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
h[4] &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
h[5] &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
h[6] &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}$$

where η_{ji} stand for $\texttt{Moments::nu}_{ji}$.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

See also: cv::matchShapes

---

cv::findContours

Finds the contours in a binary image.

```
void findContours( const Mat& image, vector<vector<Point> >& contours,
           vector<Vec4i>& hierarchy, int mode,
           int method, Point offset=Point());
void findContours( const Mat& image, vector<vector<Point> >& contours,
           int mode, int method, Point offset=Point());
```

image The source, an 8-bit single-channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary`. You can use cv::compare, cv::inRange, cv::threshold, cv::adaptiveThreshold, cv::Canny etc. to create a binary image out of a grayscale or color one. The function modifies the `image` while extracting the contours

contours The detected contours. Each contour is stored as a vector of points

hiararchy The optional output vector that will contain information about the image topology. It will have as many elements as the number of contours. For each contour `contours[i]`, the elements `hierarchy[i][0]`, `hiearchy[i][1]`, `hiearchy[i][2]`, `hiearchy[i][3]` will be set to 0-based indices in `contours` of the next and previous contours at the same hierarchical level, the first child contour and the parent contour, respectively. If for some contour `i` there is no next, previous, parent or nested contours, the corresponding elements of `hierarchy[i]` will be negative

mode The contour retrieval mode

CV_RETR_EXTERNAL retrieves only the extreme outer contours; It will set `hierarchy[i][2]=hierarchy` for all the contours

CV_RETR_LIST retrieves all of the contours without establishing any hierarchical relationships

CV_RETR_CCOMP retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes. If inside a hole of a connected component there is another contour, it will still be put on the top level

CV_RETR_TREE retrieves all of the contours and reconstructs the full hierarchy of nested contours. This full hierarchy is built and shown in OpenCV `contours.c` demo

method The contour approximation method.

CV_CHAIN_APPROX_NONE stores absolutely all the contour points. That is, every 2 points of a contour stored with this method are 8-connected neighbors of each other

CV_CHAIN_APPROX_SIMPLE compresses horizontal, vertical, and diagonal segments and leaves only their end points. E.g. an up-right rectangular contour will be encoded with 4 points

CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS applies one of the flavors of the Teh-Chin chain approximation algorithm; see [**?**]

offset The optional offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context

The function retrieves contours from the binary image using the algorithm [**?**]. The contours are a useful tool for shape analysis and object detection and recognition. See `squares.c` in the OpenCV sample directory.

---

cv::drawContours

Draws contours' outlines or filled contours.

```
void drawContours( Mat& image, const vector<vector<Point> >& contours,
           int contourIdx, const Scalar& color, int thickness=1,
           int lineType=8, const vector<Vec4i>& hierarchy=vector<Vec4i>(),
           int maxLevel=INT_MAX, Point offset=Point() );
```

image The destination image

contours All the input contours. Each contour is stored as a point vector

contourIdx Indicates the contour to draw. If it is negative, all the contours are drawn

color The contours' color

thickness Thickness of lines the contours are drawn with. If it is negative (e.g. `thickness=CV_FILLED`), the contour interiors are drawn.

lineType The line connectivity; see cv::line description

hierarchy The optional information about hierarchy. It is only needed if you want to draw only some of the contours (see `maxLevel`)

maxLevel Maximal level for drawn contours. If 0, only the specified contour is drawn. If 1, the function draws the contour(s) and all the nested contours. If 2, the function draws the contours, all the nested contours and all the nested into nested contours etc. This parameter is only taken into account when there is `hierarchy` available.

offset The optional contour shift parameter. Shift all the drawn contours by the specified $\texttt{offset} = (dx, dy)$

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$. Here is the example on how to retrieve connected components from the binary image and label them

```
#include "cv.h"
#include "highgui.h"

using namespace cv;

int main( int argc, char** argv )
{
    Mat src;
    // the first command line parameter must be file name of binary
    // (black-n-white) image
    if( argc != 2 || !(src=imread(argv[1], 0)).data)
        return -1;

    Mat dst = Mat::zeros(src.rows, src.cols, CV_8UC3);

    src = src > 1;
    namedWindow( "Source", 1 );
    imshow( "Source", src );

    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    findContours( src, contours, hierarchy,
```

```
        CV_RETR_CCOMP, CV_CHAIN_APPROX_SIMPLE );

    // iterate through all the top-level contours,
    // draw each connected component with its own random color
    int idx = 0;
    for( ; idx >= 0; idx = hiearchy[idx][0] )
    {
        Scalar color( rand()&255, rand()&255, rand()&255 );
        drawContours( dst, contours, idx, color, CV_FILLED, 8, hiearchy );
    }

    namedWindow( "Components", 1 );
    imshow( "Components", dst );
    waitKey(0);
}
```

---

cv::approxPolyDP

Approximates polygonal curve(s) with the specified precision.

```
void approxPolyDP( const Mat& curve,
           vector<Point>& approxCurve,
           double epsilon, bool closed );
void approxPolyDP( const Mat& curve,
           vector<Point2f>& approxCurve,
           double epsilon, bool closed );
```

curve The polygon or curve to approximate. Must be $1 \times N$ or $N \times 1$ matrix of type `CV_32SC2` or `CV_32FC2`. You can also convert `vector<Point>` or `vector<Point2f` to the matrix by calling `Mat(const vector<T>&)` constructor.

approxCurve The result of the approximation; The type should match the type of the input curve

epsilon Specifies the approximation accuracy. This is the maximum distance between the original curve and its approximation

closed If true, the approximated curve is closed (i.e. its first and last vertices are connected), otherwise it's not

The functions `approxPolyDP` approximate a curve or a polygon with another curve/polygon with less vertices, so that the distance between them is less or equal to the specified precision. It

used Douglas-Peucker algorithm http://en.wikipedia.org/wiki/Ramer-Douglas-Peucker_algorithm

---

cv::arcLength

Calculates a contour perimeter or a curve length.

```
double arcLength( const Mat& curve, bool closed );
```

curve The input vector of 2D points, represented by CV_32SC2 or CV_32FC2 matrix, or by vector<Point> or vector<Point2f> converted to a matrix with Mat(const vector<T>&) constructor

closed Indicates, whether the curve is closed or not

The function computes the curve length or the closed contour perimeter.

---

cv::boundingRect

Calculates the up-right bounding rectangle of a point set.

```
Rect boundingRect( const Mat& points );
```

points The input 2D point set, represented by CV_32SC2 or CV_32FC2 matrix, or by vector<Point> or vector<Point2f> converted to the matrix using Mat(const vector<T>&) constructor.

The function calculates and returns the minimal up-right bounding rectangle for the specified point set.

---

cv::estimateRigidTransform

Computes optimal affine transformation between two 2D point sets

```
Mat estimateRigidTransform( const Mat& srcpt, const Mat& dstpt,
           bool fullAffine );
```

srcpt The first input 2D point set

dst The second input 2D point set of the same size and the same type as `A`

fullAffine If true, the function finds the optimal affine transformation with no any additional resrictions (i.e. there are 6 degrees of freedom); otherwise, the class of transformations to choose from is limited to combinations of translation, rotation and uniform scaling (i.e. there are 5 degrees of freedom)

The function finds the optimal affine transform $[A|b]$ (a 2×3 floating-point matrix) that approximates best the transformation from $\texttt{srcpt}_i$ to $\texttt{dstpt}_i$:

$$[A^*|b^*] = arg \min_{[A|b]} \sum_i \|\texttt{dstpt}_i - A\texttt{srcpt}_i{}^T - b\|^2$$

where $[A|b]$ can be either arbitrary (when `fullAffine=true`) or have form

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ -a_{12} & a_{11} & b_2 \end{bmatrix}$$

when `fullAffine=false`.

See also: cv::getAffineTransform, cv::getPerspectiveTransform, cv::findHomography

---

cv::estimateAffine3D

Computes optimal affine transformation between two 3D point sets

```
int estimateAffine3D(const Mat& srcpt, const Mat& dstpt, Mat& out,
           vector<uchar>& outliers,
           double ransacThreshold = 3.0,
           double confidence = 0.99);
```

srcpt The first input 3D point set

dstpt The second input 3D point set

out The output 3D affine transformation matrix 3×4

outliers The output vector indicating which points are outliers

ransacThreshold The maximum reprojection error in RANSAC algorithm to consider a point an inlier

confidence The confidence level, between 0 and 1, with which the matrix is estimated

The function estimates the optimal 3D affine transformation between two 3D point sets using RANSAC algorithm.

---

cv::contourArea

Calculates the contour area

```
double contourArea( const Mat& contour );
```

contour The contour vertices, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

The function computes the contour area. Similarly to cv::moments the area is computed using the Green formula, thus the returned area and the number of non-zero pixels, if you draw the contour using cv::drawContours or cv::fillPoly, can be different. Here is a short example:

```
vector<Point> contour;
contour.push_back(Point2f(0, 0));
contour.push_back(Point2f(10, 0));
contour.push_back(Point2f(10, 10));
contour.push_back(Point2f(5, 4));

double area0 = contourArea(contour);
vector<Point> approx;
approxPolyDP(contour, approx, 5, true);
double area1 = contourArea(approx);

cout << "area0 =" << area0 << endl <<
        "area1 =" << area1 << endl <<
        "approx poly vertices" << approx.size() << endl;
```

---

cv::convexHull

Finds the convex hull of a point set.

```
void convexHull( const Mat& points, vector<int>& hull,
           bool clockwise=false );
void convexHull( const Mat& points, vector<Point>& hull,
           bool clockwise=false );
void convexHull( const Mat& points, vector<Point2f>& hull,
           bool clockwise=false );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

hull The output convex hull. It is either a vector of points that form the hull, or a vector of 0-based point indices of the hull points in the original array (since the set of convex hull points is a subset of the original point set).

clockwise If true, the output convex hull will be oriented clockwise, otherwise it will be oriented counter-clockwise. Here, the usual screen coordinate system is assumed - the origin is at the top-left corner, x axis is oriented to the right, and y axis is oriented downwards.

The functions find the convex hull of a 2D point set using Sklansky's algorithm [**?**] that has $O(NlogN)$ or $O(N)$ complexity (where N is the number of input points), depending on how the initial sorting is implemented (currently it is $O(NlogN)$. See the OpenCV sample `convexhull.c` that demonstrates the use of the different function variants.

---

cv::fitEllipse

Fits an ellipse around a set of 2D points.

```
RotatedRect fitEllipse( const Mat& points );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

The function calculates the ellipse that fits best (in least-squares sense) a set of 2D points. It returns the rotated rectangle in which the ellipse is inscribed.

cv::fitLine

Fits a line to a 2D or 3D point set.

```
void fitLine( const Mat& points, Vec4f& line, int distType,
           double param, double reps, double aeps );
void fitLine( const Mat& points, Vec6f& line, int distType,
           double param, double reps, double aeps );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>`, `vector<Point2f>`, `vector<Point3i>` or `vector<Point3f>` converted to the matrix by `Mat(const vector<T>&)` constructor

line The output line parameters. In the case of a 2d fitting, it is a vector of 4 floats `(vx, vy, x0, y0)` where `(vx, vy)` is a normalized vector collinear to the line and `(x0, y0)` is some point on the line. in the case of a 3D fitting it is vector of 6 floats `(vx, vy, vz, x0, y0, z0)` where `(vx, vy, vz)` is a normalized vector collinear to the line and `(x0, y0, z0)` is some point on the line

distType The distance used by the M-estimator (see the discussion)

param Numerical parameter (`C`) for some types of distances, if 0 then some optimal value is chosen

reps, aeps Sufficient accuracy for the radius (distance between the coordinate origin and the line) and angle, respectively; 0.01 would be a good default value for both.

The functions `fitLine` fit a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i^{th} point and the line and $\rho(r)$ is a distance function, one of:

distType=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

distType=CV_DIST_L1

$$\rho(r) = r$$

distType=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

distType=CV_DIST_FAIR

$$\rho(r) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right) \right) \quad \text{where} \quad C = 1.3998$$

distType=CV_DIST_WELSCH

$$\rho(r) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

distType=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

The algorithm is based on the M-estimator (http://en.wikipedia.org/wiki/M-estimator) technique, that iteratively fits the line using weighted least-squares algorithm and after each iteration the weights w_i are adjusted to beinversely proportional to $\rho(r_i)$.

---

cv::isContourConvex

Tests contour convexity.

```
bool isContourConvex( const Mat& contour );
```

contour The tested contour, a matrix of type CV_32SC2 or CV_32FC2, or vector<Point> or vector<Point2f> converted to the matrix using Mat(const vector<T>&) constructor.

The function tests whether the input contour is convex or not. The contour must be simple, i.e. without self-intersections, otherwise the function output is undefined.

---

cv::minAreaRect

Finds the minimum area rotated rectangle enclosing a 2D point set.

```
RotatedRect minAreaRect( const Mat& points );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

The function calculates and returns the minimum area bounding rectangle (possibly rotated) for the specified point set. See the OpenCV sample `minarea.c`

---

cv::minEnclosingCircle

Finds the minimum area circle enclosing a 2D point set.

```
void minEnclosingCircle( const Mat& points, Point2f& center, float&
radius );
```

points The input 2D point set, represented by `CV_32SC2` or `CV_32FC2` matrix, or by `vector<Point>` or `vector<Point2f>` converted to the matrix using `Mat(const vector<T>&)` constructor.

center The output center of the circle

radius The output radius of the circle

The function finds the minimal enclosing circle of a 2D point set using iterative algorithm. See the OpenCV sample `minarea.c`

---

cv::matchShapes

Compares two shapes.

```
double matchShapes( const Mat& object1,
           const Mat& object2,
           int method, double parameter=0 );
```

object1 The first contour or grayscale image

object2 The second contour or grayscale image

method Comparison method: `CV_CONTOUR_MATCH_I1`, `CV_CONTOURS_MATCH_I2` or `CV_CONTOURS_MATCH_I3` (see the discussion below)

parameter Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu invariants (see cv::HuMoments) as following (A denotes `object1`, B denotes `object2`):

method=CV_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1\ldots7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1\ldots7} \left| m_i^A - m_i^B \right|$$

method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1\ldots7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

where

$$m_i^A = \mathrm{sign}(h_i^A) \cdot \log h_i^A$$
$$m_i^B = \mathrm{sign}(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

---

cv::pointPolygonTest

Performs point-in-contour test.

```
double pointPolygonTest( const Mat& contour,
           Point2f pt, bool measureDist );
```

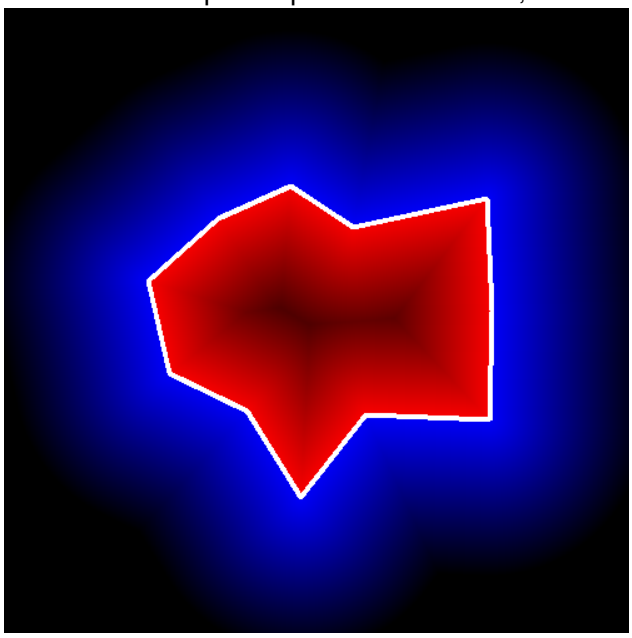
contour The input contour

pt The point tested against the contour

measureDist If true, the function estimates the signed distance from the point to the nearest contour edge; otherwise, the function only checks if the point is inside or not.

The function determines whether the point is inside a contour, outside, or lies on an edge (or coincides with a vertex). It returns positive (inside), negative (outside) or zero (on an edge) value, correspondingly. When `measureDist=false`, the return value is +1, -1 and 0, respectively. Otherwise, the return value it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



8.8 Planar Subdivisions

8.9 Object Detection

FeatureEvaluator

Base class for computing feature values in cascade classifiers

```
class FeatureEvaluator
{
public:
```

```
    // feature type
    enum { HAAR = 0, LBP = 1 };
    virtual ~FeatureEvaluator();
    // reads parameters of the features from a FileStorage node
    virtual bool read(const FileNode& node);
    // returns a full copy of the feature evaluator
    virtual Ptr<FeatureEvaluator> clone() const;
    // returns the feature type (HAAR or LBP for now)
    virtual int getFeatureType() const;

    // sets the image in which to compute the features
    // (called by CascadeClassifier::setImage)
    virtual bool setImage(const Mat& image, Size origWinSize);
    // sets window in the current image in which the features
    // will be computed (called by CascadeClassifier::runAt)
    virtual bool setWindow(Point p);

    // computes value of an ordered (numerical) feature #featureIdx
    virtual double calcOrd(int featureIdx) const;
    // computes value of a categorical feature #featureIdx
    virtual int calcCat(int featureIdx) const;

    // static function that constructs feature evaluator
    // of the specific feature type (HAAR or LBP for now)
    static Ptr<FeatureEvaluator> create(int type);
};
```

CascadeClassifier

The cascade classifier class for object detection

```
class CascadeClassifier
{
public:
    enum { BOOST = 0 };

    // default constructor
    CascadeClassifier();
    // load the classifier from file
    CascadeClassifier(const string& filename);
    // the destructor
    ~CascadeClassifier();

    // checks if the classifier has been loaded or not
```

```
    bool empty() const;
    // loads the classifier from file. The previous content is destroyed.
    bool load(const string& filename);
    // reads the classifier from a FileStorage node.
    bool read(const FileNode& node);
    // detects objects of different sizes in the input image.
    // the detected objects are returned as a list of rectangles.
    // scaleFactor specifies how much the image size
    //        is reduced at each image scale.
    // minNeighbors speficifes how many neighbors should
    //        each candiate rectangle have to retain it.
    // flags - ignored
    // minSize - the minimum possible object size.
    //        Objects smaller than that are ignored.
    void detectMultiScale( const Mat& image,
                           vector<Rect>& objects,
                           double scaleFactor=1.1,
                           int minNeighbors=3, int flags=0,
                           Size minSize=Size());
    // sets the image for detection
    // (called by detectMultiScale at each image level)
    bool setImage( Ptr<FeatureEvaluator>& feval, const Mat& image );
    // runs the detector at the specified point
    // (the image that the detector is working with should be set
    // by setImage)
    int runAt( Ptr<FeatureEvaluator>& feval, Point pt );

    bool is_stump_based;

    int stageType;
    int featureType;
    int ncategories;
    Size origWinSize;

    Ptr<FeatureEvaluator> feval;
    Ptr<CvHaarClassifierCascade> oldCascade;
};
```

---

cv::groupRectangles

Groups the object candidate rectangles

```
void groupRectangles(vector<Rect>& rectList,
           int groupThreshold, double eps=0.2);
```

rectList The input/output vector of rectangles. On output there will be retained and grouped rectangles

groupThreshold The minimum possible number of rectangles, minus 1, in a group of rectangles to retain it.

eps The relative difference between sides of the rectangles to merge them into a group

The function is a wrapper for a generic function cv::partition. It clusters all the input rectangles using the rectangle equivalence criteria, that combines rectangles that have similar sizes and similar locations (the similarity is defined by `eps`). When `eps=0`, no clustering is done at all. If `eps` $\rightarrow +\inf$, all the rectangles will be put in one cluster. Then, the small clusters, containing less than or equal to `groupThreshold` rectangles, will be rejected. In each other cluster the average rectangle will be computed and put into the output rectangle list.

---

cv::matchTemplate

Compares a template against overlapped image regions.

```
void matchTemplate( const Mat& image, const Mat& templ,
           Mat& result, int method );
```

image Image where the search is running; should be 8-bit or 32-bit floating-point

templ Searched template; must be not greater than the source image and have the same data type

result A map of comparison results; will be single-channel 32-bit floating-point. If `image` is $W \times H$ and `templ` is $w \times h$ then `result` will be $(W - w + 1) \times (H - h + 1)$

method Specifies the comparison method (see below)

The function slides through `image`, compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result`. Here are the formulas for the available comparison methods (I denotes `image`, T `template`, R `result`). The summation is done over template and/or the image patch: $x' = 0...w - 1, y' = 0...h - 1$

method=CV_TM_SQDIFF

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

method=CV_TM_SQDIFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCORR

$$R(x,y) = \sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I(x+x',y+y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'} (T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

After the function finishes the comparison, the best matches can be found as global minimums (when `CV_TM_SQDIFF` was used) or maximums (when `CV_TM_CCORR` or `CV_TM_CCOEFF` was used) using the cv::minMaxLoc function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel). That is, the function can take a color template and a color image; the result will still be a single-channel image, which is easier to analyze.

8.10 Camera Calibration and 3D Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re-used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distorion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$
where $r^2 = x'^2 + y'^2$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

k_1, k_2, k_3 are radial distortion coefficients, p_1, p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters. *And they remain the same regardless of the captured image resolution.* That is, if, for example, a camera has been calibrated on images of 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x, f_y, c_x and c_y need to be scaled appropriately).

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspodences).
- Estimate the relative position and orientation of the stereo camera ”heads” and compute the *rectification* transformation that makes the camera optical axes parallel.

CalibrateCamera2

calibrateCamera Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
double calibrateCamera( const vector<vector<Point3f> >& objectPoints,
           const vector<vector<Point2f> >& imagePoints,
           Size imageSize,
           Mat& cameraMatrix, Mat& distCoeffs,
           vector<Mat>& rvecs, vector<Mat>& tvecs,
           int flags=0 );
```

objectPoints The vector of vectors of points on the calibration pattern in its coordinate system, one vector per view. If the the same calibration pattern is shown in each view and it's fully visible then all the vectors will be the same, although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0

imagePoints The vector of vectors of the object point projections on the calibration pattern views, one vector per a view. The projections must be in the same order as the corresponding object points.

imageSize Size of the image, used only to initialize the intrinsic camera matrix

cameraMatrix The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.

If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized before calling the function

distCoeffs The output 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

rvecs The output vector of rotation vectors (see cv::Rodrigues), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0..*M*-1)

tvecs The output vector of translation vectors, estimated for each pattern view.

flags Different flags, may be 0 or combination of the following values:

CV_CALIB_USE_INTRINSIC_GUESS `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the

image center (`imageSize` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate the extrinsic parameters. Use cv::solvePnP instead.

CV_CALIB_FIX_PRINCIPAL_POINT The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.

CV_CALIB_FIX_ASPECT_RATIO The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix`.
When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see cv::findChessboardCorners). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).

2. The the initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using cv::solvePnP

3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points `imagePoints` and the projected (using the current estimates for camera parameters and the poses) object points `objectPoints`; see cv::projectPoints.

The function returns the final re-projection error.

Note: if you're using a non-square (=non-NxN) grid and cv::findChessboardCorners for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2 - 0.5, h/2 - 0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probaby used `patternSize=cvSize(rows,cols)`, but should use `patternSize=cvSize(cols,rows)` in cv::findChessboardCorners.

See also: cv::findChessboardCorners, cv::solvePnP, cv::initCameraMatrix2D, cv::stereoCalibrate, cv::undistort

---

cv::calibrationMatrixValues

Computes some useful camera characteristics from the camera matrix

```
void calibrationMatrixValues( const Mat& cameraMatrix,
           Size imageSize,
           double apertureWidth,
           double apertureHeight,
           double& fovx,
           double& fovy,
           double& focalLength,
           Point2d& principalPoint,
           double& aspectRatio );
```

cameraMatrix The input camera matrix that can be estimated by cv::calibrateCamera or cv::stereoCalibrate

imageSize The input image size in pixels

apertureWidth Physical width of the sensor

apertureHeight Physical height of the sensor

fovx The output field of view in degrees along the horizontal sensor axis

fovy The output field of view in degrees along the vertical sensor axis

focalLength The focal length of the lens in mm

prinicialPoint The principal point in pixels

aspectRatio f_y/f_x

The function computes various useful camera characteristics from the previously estimated camera matrix.

cv::composeRT

Combines two rotation-and-shift transformations

```
void composeRT( const Mat& rvec1, const Mat& tvec1,
           const Mat& rvec2, const Mat& tvec2,
           Mat& rvec3, Mat& tvec3 );
void composeRT( const Mat& rvec1, const Mat& tvec1,
           const Mat& rvec2, const Mat& tvec2,
           Mat& rvec3, Mat& tvec3,
           Mat& dr3dr1, Mat& dr3dt1,
           Mat& dr3dr2, Mat& dr3dt2,
           Mat& dt3dr1, Mat& dt3dt1,
           Mat& dt3dr2, Mat& dt3dt2 );
```

rvec1 The first rotation vector

tvec1 The first translation vector

rvec2 The second rotation vector

tvec2 The second translation vector

rvec3 The output rotation vector of the superposition

tvec3 The output translation vector of the superposition

d??d?? The optional output derivatives of `rvec3` or `tvec3` w.r.t. `rvec?` or `tvec?`

The functions compute:

$$\begin{array}{l} \texttt{rvec3} = \mathrm{rodrigues}^{-1}\left(\mathrm{rodrigues}(\texttt{rvec2}) \cdot \mathrm{rodrigues}(\texttt{rvec1})\right) \\ \texttt{tvec3} = \mathrm{rodrigues}(\texttt{rvec2}) \cdot \texttt{tvec1} + \texttt{tvec2} \end{array} ,$$

where $\mathrm{rodrigues}$ denotes a rotation vector to rotation matrix transformation, and $\mathrm{rodrigues}^{-1}$ denotes the inverse transformation, see cv::Rodrigues.

Also, the functions can compute the derivatives of the output vectors w.r.t the input vectors (see cv::matMulDeriv). The functions are used inside cv::stereoCalibrate but can also be used in your own code where Levenberg-Marquardt or another gradient-based solver is used to optimize a function that contains matrix multiplication.

ComputeCorrespondEpilines

computeCorrespondEpilines For points in one image of a stereo pair, computes the corresponding epilines in the other image.

```
void computeCorrespondEpilines( const Mat& points,
           int whichImage, const Mat& F,
           vector<Vec3f>& lines );
```

points The input points. $N \times 1$ or $1 \times N$ matrix of type `CV_32FC2` or `vector<Point2f>`

whichImage Index of the image (1 or 2) that contains the `points`

F The fundamental matrix that can be estimated using cv::findFundamentalMat or cv::stereoRectify.

lines The output vector of the corresponding to the points epipolar lines in the other image. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see cv::findFundamentalMat), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

ConvertPointsHomogeneous

convertPointsHomogeneous Convert points to/from homogeneous coordinates.

```
void convertPointsHomogeneous( const Mat& src, vector<Point3f>& dst );
           void convertPointsHomogeneous( const Mat& src,
vector<Point2f>& dst );
```

src The input array or vector of 2D or 3D points

dst The output vector of 3D or 2D points, respectively

The functions convert 2D or 3D points from/to homogeneous coordinates, or simply copy or transpose the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$x' = x/w$
$y' = y/w$
$z' = z/w$ (if output is 3D)

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

DecomposeProjectionMatrix

decomposeProjectionMatrix Decomposes the projection matrix into a rotation matrix and a camera matrix.

```
void decomposeProjectionMatrix( const Mat& projMatrix,
           Mat& cameraMatrix,
           Mat& rotMatrix, Mat& transVect );
void decomposeProjectionMatrix( const Mat& projMatrix,
           Mat& cameraMatrix,
           Mat& rotMatrix, Mat& transVect,
           Mat& rotMatrixX, Mat& rotMatrixY,
           Mat& rotMatrixZ, Vec3d& eulerAngles );
```

projMatrix The 3x4 input projection matrix P

cameraMatrix The output 3x3 camera matrix K

rotMatrix The output 3x3 external rotation matrix R

transVect The output 4x1 translation vector T

rotMatrX Optional 3x3 rotation matrix around x-axis

rotMatrY Optional 3x3 rotation matrix around y-axis

rotMatrZ Optional 3x3 rotation matrix around z-axis

eulerAngles Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on cv::RQDecomp3x3.

DrawChessboardCorners

drawChessboardCorners Renders the detected chessboard corners.

```
void drawChessboardCorners( Mat& image, Size patternSize,
           const Mat& corners,
           bool patternWasFound );
```

image The destination image; it must be an 8-bit color image

patternSize The number of inner corners per chessboard row and column. (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The array of corners detected

patternWasFound Indicates whether the complete board was found or not . One may just pass the return value cv::findChessboardCorners here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

FindChessboardCorners

findChessboardCorners Finds the positions of the internal corners of the chessboard.

```
bool findChessboardCorners( const Mat& image, Size patternSize,
           vector<Point2f>& corners,
           int flags=CV_CALIB_CB_ADAPTIVE_THRESH+
           CV_CALIB_CB_NORMALIZE_IMAGE );
```

image Source chessboard view; it must be an 8-bit grayscale or color image

patternSize The number of inner corners per chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The output array of corners detected

flags Various operation flags, can be 0 or a combination of the following values:

- **CV_CALIB_CB_ADAPTIVE_THRESH** use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
- **CV_CALIB_CB_NORMALIZE_IMAGE** normalize the image gamma with cv::equalizeHist before applying fixed or adaptive thresholding.
- **CV_CALIB_CB_FILTER_QUADS** use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cv::cornerSubPix.

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

FindExtrinsicCameraParams2

solvePnP Finds the object pose from the 3D-2D point correspondences

```
void solvePnP( const Mat& objectPoints,
           const Mat& imagePoints,
           const Mat& cameraMatrix,
           const Mat& distCoeffs,
           Mat& rvec, Mat& tvec,
           bool useExtrinsicGuess=false );
```

objectPoints The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points. Can also pass `vector<Point3f>` here.

imagePoints The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points. Can also pass `vector<Point2f>` here.

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$. If it is NULL, all of the distortion coefficients are set to 0

rvec The output rotation vector (see cv::Rodrigues) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system

tvec The output translation vector

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using cv::projectPoints) `objectPoints`.

FindFundamentalMat

findFundamentalMat Calculates the fundamental matrix from the corresponding points in two images.

```
Mat findFundamentalMat( const Mat& points1, const Mat& points2,
           vector<uchar>& status, int method=FM_RANSAC,
           double param1=3., double param2=0.99 );
Mat findFundamentalMat( const Mat& points1, const Mat& points2,
           int method=FM_RANSAC,
           double param1=3., double param2=0.99 );
```

points1 Array of `N` points from the first image. . The point coordinates should be floating-point (single or double precision)

points2 Array of the second image points of the same size and format as `points1`

method Method for computing the fundamental matrix

- **CV_FM_7POINT** for a 7-point algorithm. $N = 7$
- **CV_FM_8POINT** for an 8-point algorithm. $N \geq 8$
- **CV_FM_RANSAC** for the RANSAC algorithm. $N \geq 8$
- **CV_FM_LMEDS** for the LMedS algorithm. $N \geq 8$

param1 The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise

param2 The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct

status The output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1's

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the found fundamental matrix . Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to cv::computeCorrespondEpilines that finds the epipolar lines corresponding to the specified points. It can also be passed to cv::stereoRectifyUncalibrated to compute the rectification transformation.

```
// Example. Estimation of fundamental matrix using RANSAC algorithm
int point_count = 100;
vector<Point2f> points1(point_count);
vector<Point2f> points2(point_count);

// initialize the points here ... */
for( int i = 0; i < point_count; i++ )
{
    points1[i] = ...;
    points2[i] = ...;
```

```
}

Mat fundamental_matrix =
 findFundamentalMat(points1, points2, FM_RANSAC, 3, 0.99);
```

FindHomography

findHomography Finds the perspective transformation between two planes.

```
Mat findHomography( const Mat& srcPoints, const Mat& dstPoints,
           Mat& status, int method=0,
           double ransacReprojThreshold=0 );
Mat findHomography( const Mat& srcPoints, const Mat& dstPoints,
           vector<uchar>& status, int method=0,
           double ransacReprojThreshold=0 );
Mat findHomography( const Mat& srcPoints, const Mat& dstPoints,
           int method=0, double ransacReprojThreshold=0 );
```

srcPoints Coordinates of the points in the original plane, a matrix of type `CV_32FC2` or a `vector<Point2f>`.

dstPoints Coordinates of the points in the target plane, a matrix of type `CV_32FC2` or a `vector<Point2f>`.

method The method used to computed homography matrix; one of the following:

- **0** a regular method using all the points
- **CV_RANSAC** RANSAC-based robust method
- **CV_LMEDS** Least-Median robust method

ransacReprojThreshold The maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.

status The optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). *Note that the input mask values are ignored.*

The functions find and return the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter `method` is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, `RANSAC` and `LMeDS`, try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection error for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method `RANSAC` can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method `LMeDS` does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features, where can be only some small noise present, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized so that $h_{33} = 1$.

See also: cv::getAffineTransform, cv::getPerspectiveTransform, cv::estimateRigidMotion, cv::warpPerspective, cv::perspectiveTransform

---

cv::getDefaultNewCameraMatrix

Returns the default new camera matrix

```
Mat getDefaultNewCameraMatrix(
           const Mat& cameraMatrix,
           Size imgSize=Size(),
           bool centerPrincipalPoint=false );
```

cameraMatrix The input camera matrix

imageSize The camera view image size in pixels

centerPrincipalPoint Indicates whether in the new camera matrix the principal point should be at the image center or not

The function returns the camera matrix that is either an exact copy of the input `cameraMatrix` (when `centerPrinicipalPoint=false`), or the modified one (when `centerPrincipalPoint`=true).

In the latter case the new camera matrix will be:

$$\begin{bmatrix} f_x & 0 & (\texttt{imgSize.width} - 1) * 0.5 \\ 0 & f_y & (\texttt{imgSize.height} - 1) * 0.5 \\ 0 & 0 & 1 \end{bmatrix},$$

where f_x and f_y are $(0,0)$ and $(1,1)$ elements of `cameraMatrix`, respectively.

By default, the undistortion functions in OpenCV (see `initUndistortRectifyMap`, `undistort`) do not move the principal point. However, when you work with stereo, it's important to move the principal points in both views to the same y-coordinate (which is required by most of stereo correspondence algorithms), and maybe to the same x-coordinate too. So you can form the new camera matrix for each view, where the principal points will be at the center.

GetOptimalNewCameraMatrix

getOptimalNewCameraMatrix Returns the new camera matrix based on the free scaling parameter

```
Mat getOptimalNewCameraMatrix(
           const Mat& cameraMatrix, const Mat& distCoeffs,
           Size imageSize, double alpha, Size newImageSize=Size(),
           Rect* validPixROI=0);
```

cameraMatrix The input camera matrix

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

imageSize The original image size

alpha The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see cv::stereoRectify

newCameraMatrix The output new camera matrix.

newImageSize The image size after rectification. By default it will be set to `imageSize`.

validPixROI The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in cv::stereoRectify

The function computes and returns the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0`, keep all the original image pixels if there is valuable information in the corners `alpha=1`, or get something in between. When `alpha>0`, the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to cv::initUndistortRectifyMap to produce the maps for cv::remap.

InitIntrinsicParams2D

initCameraMatrix2D Finds the initial camera matrix from the 3D-2D point correspondences

```
Mat initCameraMatrix2D( const vector<vector<Point3f> >& objectPoints,
            const vector<vector<Point2f> >& imagePoints,
            Size imageSize, double aspectRatio=1.);
```

objectPoints The vector of vectors of the object points. See cv::calibrateCamera

imagePoints The vector of vectors of the corresponding image points. See cv::calibrateCamera

imageSize The image size in pixels; used to initialize the principal point

aspectRatio If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

InitUndistortRectifyMap

initUndistortRectifyMap Computes the undistortion and rectification transformation map.

```
void initUndistortRectifyMap( const Mat& cameraMatrix,
            const Mat& distCoeffs, const Mat& R,
            const Mat& newCameraMatrix,
            Size size, int m1type,
            Mat& map1, Mat& map2 );
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

R The optional rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv::stereoRectify can be passed here. If the matrix is empty , the identity transformation is assumed

newCameraMatrix The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$

size The undistorted image size

m1type The type of the first output map, can be `CV_32FC1` or `CV_16SC2`. See cv::convertMaps

map1 The first output map

map2 The second output map

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for cv::remap. The undistorted image will look like the original, as if it was captured with a camera with camera matrix `=newCameraMatrix` and zero distortion. In the case of monocular camera `newCameraMatrix` is usually equal to `cameraMatrix`, or it can be computed by cv::getOptimalNewCameraMatrix for a better control over scaling. In the case of stereo camera `newCameraMatrix` is normally set to `P1` or `P2` computed by cv::stereoRectify.

Also, this new camera will be oriented differently in the coordinate space, according to `R`. That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images

become horizontal and have the same y- coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by cv::remap. That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$\begin{array}{l} x \leftarrow (u - {c'}_x)/{f'}_x \\ y \leftarrow (v - {c'}_y)/{f'}_y \\ [X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\ x' \leftarrow X/W \\ y' \leftarrow Y/W \\ x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\ y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\ map_x(u, v) \leftarrow x" f_x + c_x \\ map_y(u, v) \leftarrow y" f_y + c_y \end{array}$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after cv::stereoRectify, which in its turn is called after cv::stereoCalibrate. But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using cv::stereoRectifyUncalibrated. For each camera the function computes homography `H` as the rectification transformation in pixel domain, not a rotation matrix `R` in 3D space. The `R` can be computed from `H` as

$$\texttt{R} = \texttt{cameraMatrix}^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

---

cv::matMulDeriv

Computes partial derivatives of the matrix product w.r.t each multiplied matrix

```
void matMulDeriv( const Mat& A, const Mat& B, Mat& dABdA, Mat& dABdB );
```

A The first multiplied matrix

B The second multiplied matrix

dABdA The first output derivative matrix `d(A*B)/dA` of size `A.rows*B.cols` $\times$ $A.rows * A.cols$

dABdA The second output derivative matrix `d(A*B)/dB` of size `A.rows*B.cols`$\times B.rows * B.cols$

The function computes the partial derivatives of the elements of the matrix product $A * B$ w.r.t. the elements of each of the two input matrices. The function is used to compute Jacobian matrices in cv::stereoCalibrate, but can also be used in any other similar optimization function.

ProjectPoints2

projectPoints Project 3D points on to an image plane.

```
void projectPoints( const Mat& objectPoints,
            const Mat& rvec, const Mat& tvec,
            const Mat& cameraMatrix,
            const Mat& distCoeffs,
            vector<Point2f>& imagePoints );
void projectPoints( const Mat& objectPoints,
            const Mat& rvec, const Mat& tvec,
            const Mat& cameraMatrix,
            const Mat& distCoeffs,
            vector<Point2f>& imagePoints,
            Mat& dpdrot, Mat& dpdt, Mat& dpdf,
            Mat& dpdc, Mat& dpddist,
            double aspectRatio=0 );
```

objectPoints The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel (or `vector<Point3f>`) , where N is the number of points in the view

rvec The rotation vector, see cv::Rodrigues

tvec The translation vector

cameraMatrix The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$. If it is empty , all of the distortion coefficients are considered 0's

imagePoints The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel (or `vector<Point2f>`)

dpdrot Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector

dpdt Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector

dpdf Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y

dpdc Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y

dpddist Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in cv::calibrateCamera, cv::solvePnP and cv::stereoCalibrate. The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

ReprojectImageTo3D

reprojectImageTo3D Reprojects disparity image to 3D space.

```
void reprojectImageTo3D( const Mat& disparity,
           Mat& _3dImage, const Mat& Q,
           bool handleMissingValues=false );
```

disparity The input single-channel 16-bit signed or 32-bit floating-point disparity image

_3dImage The output 3-channel floating-point image of the same size as `disparity`. Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)`, computed from the disparity map.

Q The 4×4 perspective transformation matrix that can be obtained with cv::stereoRectify

handleMissingValues If true, when the pixels with the minimal disparity (that corresponds to the ouliers; see cv::StereoBM) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \mathtt{Q} * [x\ y\ \mathtt{disparity}(x,y)\ 1]^T$$
$$\mathtt{\_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by cv::stereoRectify. To reproject a sparse set of points (x,y,d),... to 3D space, use cv::perspectiveTransform.

RQDecomp3x3

RQDecomp3x3 Computes the 'RQ' decomposition of 3x3 matrices.

```
void RQDecomp3x3( const Mat& M, Mat& R, Mat& Q );
Vec3d RQDecomp3x3( const Mat& M, Mat& R, Mat& Q,
           Mat& Qx, Mat& Qy, Mat& Qz );
```

M The 3x3 input matrix

R The output 3x3 upper-triangular matrix

Q The output 3x3 orthogonal matrix

Qx Optional 3x3 rotation matrix around x-axis

Qy Optional 3x3 rotation matrix around y-axis

Qz Optional 3x3 rotation matrix around z-axis

The function computes a RQ decomposition using the given rotations. This function is used in cv::decomposeProjectionMatrix to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles (as the return value) that could be used in OpenGL.

Rodrigues2

Rodrigues Converts a rotation matrix to a rotation vector or vice versa.

```
void Rodrigues(const Mat& src, Mat& dst);
void Rodrigues(const Mat& src, Mat& dst, Mat& jacobian);
```

src The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)

dst The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively

jacobian Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like cv::calibrateCamera, cv::stereoCalibrate or cv::solvePnP.

StereoBM

The class for computing stereo correspondence using block matching algorithm.

```
// Block matching stereo correspondence algorithm\par
class StereoBM
{
    enum { NORMALIZED_RESPONSE = CV_STEREO_BM_NORMALIZED_RESPONSE,
        BASIC_PRESET=CV_STEREO_BM_BASIC,
        FISH_EYE_PRESET=CV_STEREO_BM_FISH_EYE,
        NARROW_PRESET=CV_STEREO_BM_NARROW };
```

```
    StereoBM();
    // the preset is one of ..._PRESET above.
    // ndisparities is the size of disparity range,
    // in which the optimal disparity at each pixel is searched for.
    // SADWindowSize is the size of averaging window used to match pixel blocks
    //    (larger values mean better robustness to noise, but yield blurry disparity maps)
    StereoBM(int preset, int ndisparities=0, int SADWindowSize=21);
    // separate initialization function
    void init(int preset, int ndisparities=0, int SADWindowSize=21);
    // computes the disparity for the two rectified 8-bit single-channel images.
    // the disparity will be 16-bit signed (fixed-point) or 32-bit floating-point image of
    void operator()( const Mat& left, const Mat& right, Mat& disparity, int disptype=CV_16S

    Ptr<CvStereoBMState> state;
};
```

StereoCalibrate

stereoCalibrate Calibrates stereo camera.

```
double stereoCalibrate( const vector<vector<Point3f> >& objectPoints,
           const vector<vector<Point2f> >& imagePoints1,
           const vector<vector<Point2f> >& imagePoints2,
           Mat& cameraMatrix1, Mat& distCoeffs1,
           Mat& cameraMatrix2, Mat& distCoeffs2,
           Size imageSize, Mat& R, Mat& T,
           Mat& E, Mat& F,
           TermCriteria term_crit = TermCriteria(TermCriteria::COUNT+
           TermCriteria::EPS, 30, 1e-6),
           int flags=CALIB_FIX_INTRINSIC );
```

objectPoints The vector of vectors of points on the calibration pattern in its coordinate system, one vector per view. If the the same calibration pattern is shown in each view and it's fully visible then all the vectors will be the same, although it is possible to use partially occluded patterns, or even different patterns in different views - then the vectors will be different. The points are 3D, but since they are in the pattern coordinate system, then if the rig is planar, it may have sense to put the model to the XY coordinate plane, so that Z-coordinate of each input object point is 0

imagePoints1 The vector of vectors of the object point projections on the calibration pattern views from the 1st camera, one vector per a view. The projections must be in the same order as the corresponding object points.

imagePoints2 The vector of vectors of the object point projections on the calibration pattern views from the 2nd camera, one vector per a view. The projections must be in the same order as the corresponding object points.

cameraMatrix1 The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of `CV_CALIB_USE_INTRINSIC_GUESS`, `CV_CALIB_FIX_ASPECT_RATIO`, `CV_CALIB_FIX_INTRINSIC` or `CV_CALIB_FIX_FOCAL_LENGTH` are specified, some or all of the matrices' components must be initialized; see the flags description

distCoeffs1 The input/output lens distortion coefficients for the first camera, 4x1, 5x1, 1x4 or 1x5 floating-point vectors $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$, $j = 0, 1$. If any of `CV_CALIB_FIX_K1`, `CV_CALIB_FIX_K2` or `CV_CALIB_FIX_K3` is specified, then the corresponding elements of the distortion coefficients must be initialized.

cameraMatrix2 The input/output second camera matrix, as cameraMatrix1.

distCoeffs2 The input/output lens distortion coefficients for the second camera, as distCoeffs1.

imageSize Size of the image, used only to initialize intrinsic camera matrix.

R The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The output translation vector between the cameras' coordinate systems.

E The output essential matrix.

F The output fundamental matrix.

term_crit The termination criteria for the iterative optimization algorithm.

flags Different flags, may be 0 or combination of the following values:

- **CV_CALIB_FIX_INTRINSIC** If it is set, `cameraMatrix?`, as well as `distCoeffs?` are fixed, so that only `R, T,` `E` and `F` are estimated.
- **CV_CALIB_USE_INTRINSIC_GUESS** The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.

CV_CALIB_FIX_PRINCIPAL_POINT The principal points are fixed during the optimization.

CV_CALIB_FIX_FOCAL_LENGTH $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.

CV_CALIB_FIX_ASPECT_RATIO $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.

CV_CALIB_SAME_FOCAL_LENGTH Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$

CV_CALIB_ZERO_TANGENT_DIST Tangential distortion coefficients for each camera are set to zeros and fixed there.

CV_CALIB_FIX_K1, CV_CALIB_FIX_K2, CV_CALIB_FIX_K3 Fixes the corresponding radial distortion coefficient (the coefficient must be passed to the function)

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientatation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with cv::solvePnP), obviously, those poses will relate to each other, i.e. given (R_1, T_1) it should be possible to compute (R_2, T_2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector T: $T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform full calibration of each of the 2 cameras. However, because of the high dimensionality of the parameter space and noise in the input data the function can diverge from the correct solution. Thus, if intrinsic parameters can be estimated with high accuracy for each of the cameras individually (e.g. using cv::calibrateCamera), it is recommended to do so and then pass `CV_CALIB_FIX_INTRINSIC` flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are estimated at once, it makes sense to restrict some parameters, e.g. pass `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which are usually reasonable assumptions.

Similarly to cv::calibrateCamera, the function minimizes the total re-projection error for all the points in all the available views from both cameras. The function returns the final value of the re-projection error.

StereoRectify

stereoRectify Computes rectification transforms for each head of a calibrated stereo camera.

```
void stereoRectify( const Mat& cameraMatrix1, const Mat& distCoeffs1,
           const Mat& cameraMatrix2, const Mat& distCoeffs2,
           Size imageSize, const Mat& R, const Mat& T,
           Mat& R1, Mat& R2, Mat& P1, Mat& P2, Mat& Q,
           int flags=CALIB_ZERO_DISPARITY );
void stereoRectify( const Mat& cameraMatrix1, const Mat& distCoeffs1,
           const Mat& cameraMatrix2, const Mat& distCoeffs2,
           Size imageSize, const Mat& R, const Mat& T,
           Mat& R1, Mat& R2, Mat& P1, Mat& P2, Mat& Q,
           double alpha, Size newImageSize=Size(),
           Rect* roi1=0, Rect* roi2=0,
           int flags=CALIB_ZERO_DISPARITY );
```

cameraMatrix1, cameraMatrix2 The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs1, distCoeffs2 The input distortion coefficients for each camera, ${k_1}^{(j)}, {k_2}^{(j)}, {p_1}^{(j)}, {p_2}^{(j)}[, {k_3}^{(j)}]$

imageSize Size of the image used for stereo calibration.

R The rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The translation vector between the cameras' coordinate systems.

R1, R2 The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.

P1, P2 The output 3×4 projection matrices in the new (rectified) coordinate systems.

Q The output 4×4 disparity-to-depth mapping matrix, see cv::reprojectImageTo3D.

flags The operation flags; may be 0 or CV_CALIB_ZERO_DISPARITY. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.

alpha The free scaling parameter. If it is -1 or absent , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.

newImageSize The new image resolution after rectification. The same size should be passed to cv::initUndistortRectifyMap, see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.

roi1, roi2 The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0`, the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by cv::stereoCalibrate and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

1. Horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

   where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. Vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines

in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

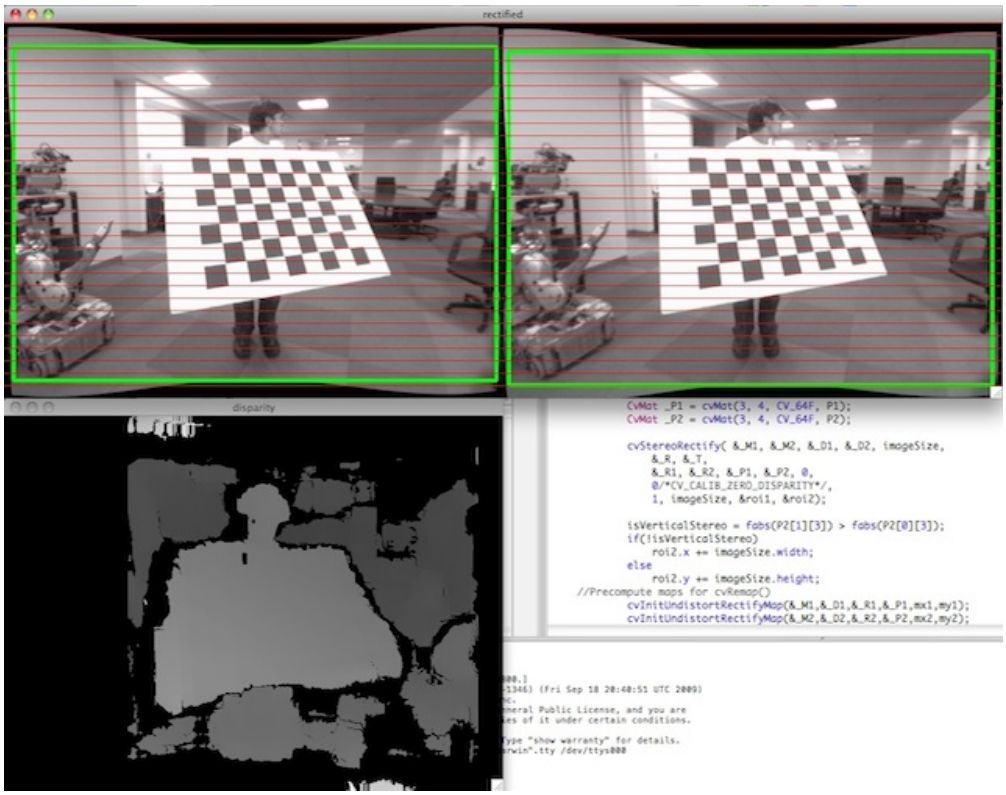
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of `P1` and `P2` will effectively be the new "rectified" camera matrices. The matrices, together with `R1` and `R2`, can then be passed to cv::initUndistortRectifyMap to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



StereoRectifyUncalibrated

stereoRectifyUncalibrated Computes rectification transform for uncalibrated stereo camera.

```
bool stereoRectifyUncalibrated( const Mat& points1,
            const Mat& points2,
            const Mat& F, Size imgSize,
            Mat& H1, Mat& H2,
            double threshold=5 );
```

points1, points2 The 2 arrays of corresponding 2D points. The same formats as in cv::findFundamentalMat are supported

F The input fundamental matrix. It can be computed from the same set of point pairs using cv::findFundamentalMat.

imageSize Size of the image.

H1, H2 The output rectification homography matrices for the first and for the second images.

threshold The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from cv::stereoRectify is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the algorithm [?].

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using cv::calibrateCamera and then the images can be corrected using cv::undistort, or just the point coordinates can be corrected with cv::undistortPoints.

Undistort2

undistort Transforms an image to compensate for lens distortion.

```
void undistort( const Mat& src, Mat& dst, const Mat& cameraMatrix,
           const Mat& distCoeffs, const Mat& newCameraMatrix=Mat() );
```

src The input (distorted) image

dst The output (corrected) image; will have the same size and the same type as `src`

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The vector of distortion coefficients, $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$

newCameraMatrix Camera matrix of the distorted image. By default it is the same as `cameraMatrix`, but you may additionally scale and shift the result by using some different matrix

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of cv::initUndistortRectifyMap (with unity R) and cv::remap (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by newCameraMatrix. You can use cv::getOptimalNewCameraMatrix to compute the appropriate newCameraMatrix, depending on your requirements.

The camera matrix and the distortion parameters can be determined using cv::calibrateCamera. If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

UndistortPoints

undistortPoints Computes the ideal point coordinates from the observed point coordinates.

```
void undistortPoints( const Mat& src, vector<Point2f>& dst,
           const Mat& cameraMatrix, const Mat& distCoeffs,
           const Mat& R=Mat(), const Mat& P=Mat());
void undistortPoints( const Mat& src, Mat& dst,
           const Mat& cameraMatrix, const Mat& distCoeffs,
           const Mat& R=Mat(), const Mat& P=Mat());
```

src The observed point coordinates, same format as imagePoints in cv::projectPoints

dst The output ideal point coordinates, after undistortion and reverse perspective transformation .

cameraMatrix The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The vector of distortion coefficients, $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$

R The rectification transformation in object space (3x3 matrix). R1 or R2, computed by cv::StereoRectify can be passed here. If the matrix is empty, the identity transformation is used

P The new camera matrix (3x3) or the new projection matrix (3x4). P1 or P2, computed by cv::StereoRectify can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to cv::undistort and cv::initUndistortRectifyMap, but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to cv::projectPoints (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used both for a stereo camera head or for monocular camera (when R is empty).

Chapter 9

cvaux. Extra Computer Vision Functionality

Chapter 10

highgui. High-level GUI and Media I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and ”remember” their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

10.1 User Interface

---

cv::createTrackbar

Creates a trackbar and attaches it to the specified window

```
int createTrackbar( const string& trackbarname,
           const string& winname,
           int* value, int count,
           TrackbarCallback onChange CV_DEFAULT(0),
           void* userdata CV_DEFAULT(0));
```

trackbarname Name of the created trackbar.

winname Name of the window which will be used as a parent of the created trackbar.

value The optional pointer to an integer variable, whose value will reflect the position of the slider. Upon creation, the slider position is defined by this variable.

count The maximal position of the slider. The minimal position is always 0.

onChange Pointer to the function to be called every time the slider changes position. This function should be prototyped as `void Foo(int,void*);`, where the first parameter is the trackbar position and the second parameter is the user data (see the next parameter). If the callback is NULL pointer, then no callbacks is called, but only `value` is updated

userdata The user data that is passed as-is to the callback; it can be used to handle trackbar events without using global variables

The function `createTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable `value` to be syncronized with trackbar position and specifies a callback function `onChange` to be called on the trackbar position change. The created trackbar is displayed on the top of the given window.

---

cv::getTrackbarPos

Returns the trackbar position.

```
int getTrackbarPos( const string& trackbarname,
           const string& winname );
```

trackbarname Name of the trackbar.

winname Name of the window which is the parent of the trackbar.

The function returns the current position of the specified trackbar.

---

cv::imshow

Displays the image in the specified window

```
void imshow( const string& winname,
             const Mat& image );
```

winname Name of the window.

image Image to be shown.

The function imshow displays the image in the specified window. If the window was created with the CV_WINDOW_AUTOSIZE flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

---

cv::namedWindow

Creates a window.

```
void namedWindow( const string& winname,
             int flags );
```

name Name of the window in the window caption that may be used as a window identifier.

flags Flags of the window. Currently the only supported flag is CV_WINDOW_AUTOSIZE. If this is set, the window size is automatically adjusted to fit the displayed image (see imshow), and the user can not change the window size manually.

The function namedWindow creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing.

cv::setTrackbarPos

Sets the trackbar position.

```
void setTrackbarPos( const string& trackbarname,
           const string& winname, int pos );
```

trackbarname Name of the trackbar.

winname Name of the window which is the parent of trackbar.

pos The new position.

The function sets the position of the specified trackbar in the specified window.

cv::waitKey

Waits for a pressed key.

```
int waitKey(int delay=0);
```

delay Delay in milliseconds. 0 is the special value that means ”forever”

The function `waitKey` waits for key event infinitely (when `delay` ≤ 0) or for `delay` milliseconds, when it's positive. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

10.2 Reading and Writing Images and Video

cv::imdecode

Reads an image from a buffer in memory.

```
Mat imdecode( const Mat& buf,
              int flags );
```

buf The input array of vector of bytes

flags The same flags as in imread

The function reads image from the specified buffer in memory. If the buffer is too short or contains invalid data, the empty matrix will be returned.

See imread for the list of supported formats and the flags description.

---

cv::imencode

Encode an image into a memory buffer.

```
bool imencode( const string& ext,
               const Mat& img,
               vector<uchar>& buf,
               const vector<int>& params=vector<int>());
```

ext The file extension that defines the output format

img The image to be written

buf The output buffer; resized to fit the compressed image

params The format-specific parameters; see imwrite

The function compresses the image and stores it in the memory buffer, which is resized to fit the result. See imwrite for the list of supported formats and the flags description.

---

cv::imread

Loads an image from a file.

```
Mat imread( const string& filename,
            int flags=1 );
```

filename Name of file to be loaded.

flags Specifies color type of the loaded image:

- **>0** the loaded image is forced to be a 3-channel color image
- **=0** the loaded image is forced to be grayscale
- **<0** the loaded image will be loaded as-is (note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB if $flags \geq 0$).

The function `imread` loads an image from the specified file and returns it. If the image can not be read (because of missing file, improper permissions, unsupported or invalid format), the function returns empty matrix (`Mat::data==NULL`).Currently, the following file formats are supported:

- Windows bitmaps - `*.bmp, *.dib` (always supported)
- JPEG files - `*.jpeg, *.jpg, *.jpe` (see **Note2**)
- JPEG 2000 files - `*.jp2` (see **Note2**)
- Portable Network Graphics - `*.png` (see **Note2**)
- Portable image format - `*.pbm, *.pgm, *.ppm` (always supported)
- Sun rasters - `*.sr, *.ras` (always supported)
- TIFF files - `*.tiff, *.tif` (see **Note2**)

Note1: The function determines type of the image by the content, not by the file extension.

Note2: On Windows and MacOSX the shipped with OpenCV image codecs (libjpeg, libpng, libtiff and libjasper) are used by default; so OpenCV can always read JPEGs, PNGs and TIFFs. On MacOSX there is also the option to use native MacOSX image readers. But beware that currently these native image loaders give images with somewhat different pixel values, because of the embedded into MacOSX color management.

On Linux, BSD flavors and other Unix-like open-source operating systems OpenCV looks for the supplied with OS image codecs. Please, install the relevant packages (do not forget the development files, e.g. ”libjpeg-dev” etc. in Debian and Ubuntu) in order to get the codec support, or turn on `OPENCV_BUILD_3RDPARTY_LIBS` flag in CMake.

cv::imwrite

Saves an image to a specified file.

```
bool imwrite( const string& filename,
              const Mat& img,
              const vector<int>& params=vector<int>());
```

filename Name of the file.

img The image to be saved.

params The format-specific save parameters, encoded as pairs `paramId_1, paramValue_1, paramId_2, paramValue_2, ...`. The following parameters are currently supported:

- In the case of JPEG it can be a quality (`CV_IMWRITE_JPEG_QUALITY`), from 0 to 100 (the higher is the better), 95 by default.
- In the case of PNG it can be the compression level (`CV_IMWRITE_PNG_COMPRESSION`), from 0 to 9 (the higher value means smaller size and longer compression time), 3 by default.
- In the case of PPM, PGM or PBM it can a binary format flag (`CV_IMWRITE_PXM_BINARY`), 0 or 1, 1 by default.

The function `imwrite` saves the image to the specified file. The image format is chosen based on the `filename` extension, see imread for the list of extensions. Only 8-bit (or 16-bit in the case of PNG, JPEG 2000 and TIFF) single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use Mat::convertTo , and cvtColor to convert it before saving, or use the universal XML I/O functions to save the image to XML or YAML format.

VideoCapture

Class for video capturing from video files or cameras

```
class VideoCapture
{
public:
    // the default constructor
    VideoCapture();
    // the constructor that opens video file
```

```
    VideoCapture(const string& filename);
    // the constructor that starts streaming from the camera
    VideoCapture(int device);

    // the destructor
    virtual ~VideoCapture();

    // opens the specified video file
    virtual bool open(const string& filename);

    // starts streaming from the specified camera by its id
    virtual bool open(int device);

    // returns true if the file was open successfully or if the camera
    // has been initialized succesfully
    virtual bool isOpened() const;

    // closes the camera stream or the video file
    // (automatically called by the destructor)
    virtual void release();

    // grab the next frame or a set of frames from a multi-head camera;
    // returns false if there are no more frames
    virtual bool grab();
    // reads the frame from the specified video stream
    // (non-zero channel is only valid for multi-head camera live streams)
    virtual bool retrieve(Mat& image, int channel=0);
    // equivalent to grab() + retrieve(image, 0);
    virtual VideoCapture& operator >> (Mat& image);

    // sets the specified property propId to the specified value
    virtual bool set(int propId, double value);
    // retrieves value of the specified property
    virtual double get(int propId);

protected:
    ...
};
```

The class provides C++ video capturing API. Here is how the class can be used:

```
#include "cv.h"
#include "highgui.h"

using namespace cv;
```

```
int main(int, char**)
{
    VideoCapture cap(0); // open the default camera
    if(!cap.isOpened())  // check if we succeeded
        return -1;

    Mat edges;
    namedWindow("edges",1);
    for(;;)
    {
        Mat frame;
        cap >> frame; // get a new frame from camera
        cvtColor(frame, edges, CV_BGR2GRAY);
        GaussianBlur(edges, edges, Size(7,7), 1.5, 1.5);
        Canny(edges, edges, 0, 30, 3);
        imshow("edges", edges);
        if(waitKey(30) >= 0) break;
    }
    // the camera will be deinitialized automatically in VideoCapture destructor
    return 0;
}
```

VideoWriter

Video writer class

```
class VideoWriter
{
public:
    // default constructor
    VideoWriter();
    // constructor that calls open
    VideoWriter(const string& filename, int fourcc,
                double fps, Size frameSize, bool isColor=true);

    // the destructor
    virtual ~VideoWriter();

    // opens the file and initializes the video writer.
    // filename - the output file name.
    // fourcc - the codec
    // fps - the number of frames per second
    // frameSize - the video frame size
    // isColor - specifies whether the video stream is color or grayscale
```

```
    virtual bool open(const string& filename, int fourcc,
                      double fps, Size frameSize, bool isColor=true);

    // returns true if the writer has been initialized successfully
    virtual bool isOpened() const;

    // writes the next video frame to the stream
    virtual VideoWriter& operator << (const Mat& image);

protected:
    ...
};
```

Chapter 11

ml. Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different seta of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class 'CvStatModel' that all the other ML classes are derived from.

11.1 Statistical Models

CvStatModel

Base class for the statistical models in ML.

```
class CvStatModel
{
public:
    /* CvStatModel(); */
    /* CvStatModel( const CvMat* train_data ... ); */

    virtual ~CvStatModel();

    virtual void clear()=0;

    /* virtual bool train( const CvMat* train_data, [int tflag,] ..., const
        CvMat* responses, ...,
     [const CvMat* var_idx,] ..., [const CvMat* sample_idx,] ...
     [const CvMat* var_type,] ..., [const CvMat* missing_mask,]
```

```
        <misc_training_alg_params> ... )=0;
      */

    /* virtual float predict( const CvMat* sample ... ) const=0; */

    virtual void save( const char* filename, const char* name=0 )=0;
    virtual void load( const char* filename, const char* name=0 )=0;

    virtual void write( CvFileStorage* storage, const char* name )=0;
    virtual void read( CvFileStorage* storage, CvFileNode* node )=0;
};
```

In this declaration some methods are commented off. Actually, these are methods for which there is no unified API (with the exception of the default constructor), however, there are many similarities in the syntax and semantics that are briefly described below in this section, as if they are a part of the base class.

CvStatModel::CvStatModel

Default constructor.

```
CvStatModel::CvStatModel();
```

Each statistical model class in ML has a default constructor without parameters. This constructor is useful for 2-stage model construction, when the default constructor is followed by `train()` or `load()`.

CvStatModel::CvStatModel(...)

Training constructor.

```
CvStatModel::CvStatModel( const CvMat* train_data ... );
```

Most ML classes provide single-step construct and train constructors. This constructor is equivalent to the default constructor, followed by the `train()` method with the parameters that are passed to the constructor.

CvStatModel:: CvStatModel

Virtual destructor.

```
CvStatModel:: CvStatModel();
```

The destructor of the base class is declared as virtual, so it is safe to write the following code:

```
CvStatModel* model;
if( use\_svm )
    model = new CvSVM(... /* SVM params */);
else
    model = new CvDTree(... /* Decision tree params */);
...
delete model;
```

Normally, the destructor of each derived class does nothing, but in this instance it calls the overridden method `clear()` that deallocates all the memory.

CvStatModel::clear

Deallocates memory and resets the model state.

```
void CvStatModel::clear();
```

The method `clear` does the same job as the destructor; it deallocates all the memory occupied by the class members. But the object itself is not destructed, and can be reused further. This method is called from the destructor, from the `train` methods of the derived classes, from the methods `load()`, `read()` or even explicitly by the user.

CvStatModel::save

Saves the model to a file.

```
void CvStatModel::save( const char* filename, const char* name=0 );
```

The method `save` stores the complete model state to the specified XML or YAML file with the specified name or default name (that depends on the particular class). `Data persistence` functionality from CxCore is used.

CvStatModel::load

Loads the model from a file.

```
void CvStatModel::load( const char* filename, const char* name=0 );
```

The method `load` loads the complete model state with the specified name (or default model-dependent name) from the specified XML or YAML file. The previous model state is cleared by `clear()`.

Note that the method is virtual, so any model can be loaded using this virtual method. However, unlike the C types of OpenCV that can be loaded using the generic crosscvLoad, here the model type must be known, because an empty model must be constructed beforehand. This limitation will be removed in the later ML versions.

CvStatModel::write

Writes the model to file storage.

```
void CvStatModel::write( CvFileStorage* storage, const char* name );
```

The method `write` stores the complete model state to the file storage with the specified name or default name (that depends on the particular class). The method is called by `save()`.

CvStatModel::read

Reads the model from file storage.

```
void CvStatMode::read( CvFileStorage* storage, CvFileNode* node );
```

The method `read` restores the complete model state from the specified node of the file storage. The node must be located by the user using the function GetFileNodeByName .

The previous model state is cleared by `clear()`.

CvStatModel::train

Trains the model.

```
bool CvStatMode::train( const CvMat* train_data, [int tflag,] ...,
const CvMat* responses, ...,
          [const CvMat* var_idx,] ..., [const CvMat* sample_idx,] ...
          [const CvMat* var_type,] ..., [const CvMat* missing_mask,]
<misc_training_alg_params> ... );
```

The method trains the statistical model using a set of input feature vectors and the corresponding output values (responses). Both input and output vectors/values are passed as matrices. By default the input feature vectors are stored as `train_data` rows, i.e. all the components (features) of a training vector are stored continuously. However, some algorithms can handle the transposed representation, when all values of each particular feature (component/input variable) over the whole input set are stored continuously. If both layouts are supported, the method includes `tflag` parameter that specifies the orientation:

- `tflag=CV_ROW_SAMPLE` means that the feature vectors are stored as rows,
- `tflag=CV_COL_SAMPLE` means that the feature vectors are stored as columns.

The `train_data` must have a `CV_32FC1` (32-bit floating-point, single-channel) format. Responses are usually stored in the 1d vector (a row or a column) of `CV_32SC1` (only in the classification problem) or `CV_32FC1` format, one value per input vector (although some algorithms, like various flavors of neural nets, take vector responses).

For classification problems the responses are discrete class labels; for regression problems the responses are values of the function to be approximated. Some algorithms can deal only with classification problems, some - only with regression problems, and some can deal with both problems. In the latter case the type of output variable is either passed as separate parameter, or as a last element of `var_type` vector:

- `CV_VAR_CATEGORICAL` means that the output values are discrete class labels,
- `CV_VAR_ORDERED(=CV_VAR_NUMERICAL)` means that the output values are ordered, i.e. 2 different values can be compared as numbers, and this is a regression problem

The types of input variables can be also specified using `var_type`. Most algorithms can handle only ordered input variables.

Many models in the ML may be trained on a selected feature subset, and/or on a selected sample subset of the training set. To make it easier for the user, the method `train` usually includes

var_idx and sample_idx parameters. The former identifies variables (features) of interest, and the latter identifies samples of interest. Both vectors are either integer (CV_32SC1) vectors, i.e. lists of 0-based indices, or 8-bit (CV_8UC1) masks of active variables/samples. The user may pass NULL pointers instead of either of the arguments, meaning that all of the variables/samples are used for training.

Additionally some algorithms can handle missing measurements, that is when certain features of certain training samples have unknown values (for example, they forgot to measure a temperature of patient A on Monday). The parameter missing_mask, an 8-bit matrix the same size as train_data, is used to mark the missed values (non-zero elements of the mask).

Usually, the previous model state is cleared by clear() before running the training procedure. However, some algorithms may optionally update the model state with the new training data, instead of resetting it.

CvStatModel::predict

Predicts the response for the sample.

```
float CvStatMode::predict( const CvMat* sample[, <prediction_params>] )
const;
```

The method is used to predict the response for a new sample. In the case of classification the method returns the class label, in the case of regression - the output function value. The input sample must have as many components as the train_data passed to train contains. If the var_idx parameter is passed to train, it is remembered and then is used to extract only the necessary components from the input sample in the method predict.

The suffix ”const” means that prediction does not affect the internal model state, so the method can be safely called from within different threads.

11.2 Normal Bayes Classifier

This is a simple classification model assuming that feature vectors from each class are normally distributed (though, not necessarily independently distributed), so the whole data distribution function is assumed to be a Gaussian mixture, one component per class. Using the training data the algorithm estimates mean vectors and covariance matrices for every class, and then it uses them for prediction.

[Fukunaga90] K. Fukunaga. Introduction to Statistical Pattern Recognition. second ed., New York: Academic Press, 1990.

CvNormalBayesClassifier

Bayes classifier for normally distributed data.

```
class CvNormalBayesClassifier : public CvStatModel
{
public:
    CvNormalBayesClassifier();
    virtual ~CvNormalBayesClassifier();

    CvNormalBayesClassifier( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx=0, const CvMat* _sample_idx=0 );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx = 0, const CvMat* _sample_idx=0, bool update=false );

    virtual float predict( const CvMat* _samples, CvMat* results=0 ) const;
    virtual void clear();

    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );
protected:
    ...
};
```

CvNormalBayesClassifier::train

Trains the model.

```
bool CvNormalBayesClassifier::train(
           const CvMat* _train_data,
           const CvMat* _responses,
           const CvMat* _var_idx =0,
           const CvMat* _sample_idx=0,
           bool update=false );
```

The method trains the Normal Bayes classifier. It follows the conventions of the generic `train` "method" with the following limitations: only CV_ROW_SAMPLE data layout is supported; the input

variables are all ordered; the output variable is categorical (i.e. elements of _responses must be integer numbers, though the vector may have CV_32FC1 type), and missing measurements are not supported.

In addition, there is an update flag that identifies whether the model should be trained from scratch (update=false) or should be updated using the new training data (update=true).

CvNormalBayesClassifier::predict

Predicts the response for sample(s)

```
float CvNormalBayesClassifier::predict(
           const CvMat* samples,
           CvMat* results=0 ) const;
```

The method predict estimates the most probable classes for the input vectors. The input vectors (one or more) are stored as rows of the matrix samples. In the case of multiple input vectors, there should be one output vector results. The predicted class for a single input vector is returned by the method.

11.3 K Nearest Neighbors

The algorithm caches all of the training samples, and predicts the response for a new sample by analyzing a certain number (**K**) of the nearest neighbors of the sample (using voting, calculating weighted sum etc.) The method is sometimes referred to as ”learning by example”, because for prediction it looks for the feature vector with a known response that is closest to the given vector.

CvKNearest

K Nearest Neighbors model.

```
class CvKNearest : public CvStatModel
{
public:

    CvKNearest();
    virtual ~CvKNearest();

    CvKNearest( const CvMat* _train_data, const CvMat* _responses,
                const CvMat* _sample_idx=0, bool _is_regression=false, int max_k=32 );
```

```
    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
                        const CvMat* _sample_idx=0, bool is_regression=false,
                        int _max_k=32, bool _update_base=false );

    virtual float find_nearest( const CvMat* _samples, int k, CvMat* results,
        const float** neighbors=0, CvMat* neighbor_responses=0, CvMat* dist=0 ) const;

    virtual void clear();
    int get_max_k() const;
    int get_var_count() const;
    int get_sample_count() const;
    bool is_regression() const;

protected:
    ...
};
```

CvKNearest::train

Trains the model.

```
bool CvKNearest::train(
            const CvMat* _train_data,
            const CvMat* _responses,
            const CvMat* _sample_idx=0,
            bool is_regression=false,
            int _max_k=32,
            bool _update_base=false );
```

The method trains the K-Nearest model. It follows the conventions of generic `train` "method" with the following limitations: only CV_ROW_SAMPLE data layout is supported, the input variables are all ordered, the output variables can be either categorical (`is_regression=false`) or ordered (`is_regression=true`), variable subsets (`var_idx`) and missing measurements are not supported.

The parameter `_max_k` specifies the number of maximum neighbors that may be passed to the method `find_nearest`.

The parameter `_update_base` specifies whether the model is trained from scratch (`_update_base=false`), or it is updated using the new training data (`_update_base=true`). In the latter case the parameter `_max_k` must not be larger than the original value.

CvKNearest::find_nearest

Finds the neighbors for the input vectors.

```
float CvKNearest::find_nearest(
           const CvMat* _samples,
           int k, CvMat* results=0,
           const float** neighbors=0,
           CvMat* neighbor_responses=0,
           CvMat* dist=0 ) const;
```

For each input vector (which are the rows of the matrix `_samples`) the method finds the $k \leq$ `get_max_k()` nearest neighbor. In the case of regression, the predicted result will be a mean value of the particular vector's neighbor responses. In the case of classification the class is determined by voting.

For custom classification/regression prediction, the method can optionally return pointers to the neighbor vectors themselves (`neighbors`, an array of `k*_samples->rows` pointers), their corresponding output values (`neighbor_responses`, a vector of `k*_samples->rows` elements) and the distances from the input vectors to the neighbors (`dist`, also a vector of `k*_samples->rows` elements).

For each input vector the neighbors are sorted by their distances to the vector.

If only a single input vector is passed, all output matrices are optional and the predicted value is returned by the method.

```
#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    const int K = 10;
    int i, j, k, accuracy;
    float response;
    int train_sample_count = 100;
    CvRNG rng_state = cvRNG(-1);
    CvMat* trainData = cvCreateMat( train_sample_count, 2, CV_32FC1 );
    CvMat* trainClasses = cvCreateMat( train_sample_count, 1, CV_32FC1 );
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    float _sample[2];
    CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
    cvZero( img );
```

```
    CvMat trainData1, trainData2, trainClasses1, trainClasses2;

    // form the training samples
    cvGetRows( trainData, &trainData1, 0, train_sample_count/2 );
    cvRandArr( &rng_state, &trainData1, CV_RAND_NORMAL, cvScalar(200,200), cvScalar(50,50)

    cvGetRows( trainData, &trainData2, train_sample_count/2, train_sample_count );
    cvRandArr( &rng_state, &trainData2, CV_RAND_NORMAL, cvScalar(300,300), cvScalar(50,50)

    cvGetRows( trainClasses, &trainClasses1, 0, train_sample_count/2 );
    cvSet( &trainClasses1, cvScalar(1) );

    cvGetRows( trainClasses, &trainClasses2, train_sample_count/2, train_sample_count );
    cvSet( &trainClasses2, cvScalar(2) );

    // learn classifier
    CvKNearest knn( trainData, trainClasses, 0, false, K );
    CvMat* nearests = cvCreateMat( 1, K, CV_32FC1);

    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;

            // estimates the response and get the neighbors' labels
            response = knn.find_nearest(&sample,K,0,0,nearests,0);

            // compute the number of neighbors representing the majority
            for( k = 0, accuracy = 0; k < K; k++ )
            {
                if( nearests->data.fl[k] == response)
                    accuracy++;
            }
            // highlight the pixel depending on the accuracy (or confidence)
            cvSet2D( img, i, j, response == 1 ?
                (accuracy > 5 ? CV_RGB(180,0,0) : CV_RGB(180,120,0)) :
                (accuracy > 5 ? CV_RGB(0,180,0) : CV_RGB(120,120,0)) );
        }
    }

    // display the original training samples
    for( i = 0; i < train_sample_count/2; i++ )
    {
```

```
        CvPoint pt;
        pt.x = cvRound(trainData1.data.fl[i*2]);
        pt.y = cvRound(trainData1.data.fl[i*2+1]);
        cvCircle( img, pt, 2, CV_RGB(255,0,0), CV_FILLED );
        pt.x = cvRound(trainData2.data.fl[i*2]);
        pt.y = cvRound(trainData2.data.fl[i*2+1]);
        cvCircle( img, pt, 2, CV_RGB(0,255,0), CV_FILLED );
    }

    cvNamedWindow( "classifier result", 1 );
    cvShowImage( "classifier result", img );
    cvWaitKey(0);

    cvReleaseMat( &trainClasses );
    cvReleaseMat( &trainData );
    return 0;
}
```

11.4 Support Vector Machines

Originally, support vector machines (SVM) was a technique for building an optimal (in some sense) binary (2-class) classifier. Then the technique has been extended to regression and clustering problems. SVM is a partial case of kernel-based methods, it maps feature vectors into higher-dimensional space using some kernel function, and then it builds an optimal linear discriminating function in this space (or an optimal hyper-plane that fits into the training data, ...). in the case of SVM the kernel is not defined explicitly. Instead, a distance between any 2 points in the hyper-space needs to be defined.

The solution is optimal in a sense that the margin between the separating hyper-plane and the nearest feature vectors from the both classes (in the case of 2-class classifier) is maximal. The feature vectors that are the closest to the hyper-plane are called ”support vectors”, meaning that the position of other vectors does not affect the hyper-plane (the decision function).

There are a lot of good references on SVM. Here are only a few ones to start with.

- **[Burges98] C. Burges. ”A tutorial on support vector machines for pattern recognition”, Knowledge Discovery and Data Mining 2(2), 1998.** (available online at http://citeseer.ist.psu.edu/burges98tutorial.html).

- **LIBSVM - A Library for Support Vector Machines. By Chih-Chung Chang and Chih-Jen Lin** (http://www.csie.ntu.edu.tw/~cjlin/libsvm/)

CvSVM

Support Vector Machines.

```
class CvSVM : public CvStatModel
{
public:
    // SVM type
    enum { C_SVC=100, NU_SVC=101, ONE_CLASS=102, EPS_SVR=103, NU_SVR=104 };

    // SVM kernel type
    enum { LINEAR=0, POLY=1, RBF=2, SIGMOID=3 };

    // SVM params type
    enum { C=0, GAMMA=1, P=2, NU=3, COEF=4, DEGREE=5 };

    CvSVM();
    virtual ~CvSVM();

    CvSVM( const CvMat* _train_data, const CvMat* _responses,
           const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
           CvSVMParams _params=CvSVMParams() );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
                        const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
                        CvSVMParams _params=CvSVMParams() );

    virtual bool train_auto( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx, const CvMat* _sample_idx, CvSVMParams _params,
        int k_fold = 10,
        CvParamGrid C_grid      = get_default_grid(CvSVM::C),
        CvParamGrid gamma_grid  = get_default_grid(CvSVM::GAMMA),
        CvParamGrid p_grid      = get_default_grid(CvSVM::P),
        CvParamGrid nu_grid     = get_default_grid(CvSVM::NU),
        CvParamGrid coef_grid   = get_default_grid(CvSVM::COEF),
        CvParamGrid degree_grid = get_default_grid(CvSVM::DEGREE) );

    virtual float predict( const CvMat* _sample ) const;
    virtual int get_support_vector_count() const;
    virtual const float* get_support_vector(int i) const;
    virtual CvSVMParams get_params() const { return params; };
    virtual void clear();

    static CvParamGrid get_default_grid( int param_id );
```

```
    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );
    int get_var_count() const { return var_idx ? var_idx->cols : var_all; }

protected:
    ...
};
```

CvSVMParams

SVM training parameters.

```
struct CvSVMParams
{
    CvSVMParams();
    CvSVMParams( int _svm_type, int _kernel_type,
                 double _degree, double _gamma, double _coef0,
                 double _C, double _nu, double _p,
                 CvMat* _class_weights, CvTermCriteria _term_crit );

    int         svm_type;
    int         kernel_type;
    double      degree; // for poly
    double      gamma;  // for poly/rbf/sigmoid
    double      coef0;  // for poly/sigmoid

    double      C;  // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
    double      nu; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
    double      p; // for CV_SVM_EPS_SVR
    CvMat*      class_weights; // for CV_SVM_C_SVC
    CvTermCriteria term_crit; // termination criteria
};
```

The structure must be initialized and passed to the training method of CvSVM .

CvSVM::train

Trains SVM.

```
bool CvSVM::train(
            const CvMat* _train_data,
            const CvMat* _responses,
            const CvMat* _var_idx=0,
            const CvMat* _sample_idx=0,
            CvSVMParams _params=CvSVMParams() );
```

The method trains the SVM model. It follows the conventions of the generic `train` "method" with the following limitations: only the CV_ROW_SAMPLE data layout is supported, the input variables are all ordered, the output variables can be either categorical (`_params.svm_type=CvSVM::C_SVC` or `_params.svm_type=CvSVM::NU_SVC`), or ordered (`_params.svm_type=CvSVM::EPS_SVR` or `_params.svm_type=CvSVM::NU_SVR`), or not required at all (`_params.svm_type=CvSVM::ONE_CLASS`), missing measurements are not supported.

All the other parameters are gathered in CvSVMParams structure.

CvSVM::train_auto

Trains SVM with optimal parameters.

```
train_auto(
            const CvMat* _train_data,
            const CvMat* _responses,
            const CvMat* _var_idx,
            const CvMat* _sample_idx,
            CvSVMParams params,
            int k_fold = 10,
            CvParamGrid C_grid = get_default_grid(CvSVM::C),
            CvParamGrid gamma_grid = get_default_grid(CvSVM::GAMMA),
            CvParamGrid p_grid = get_default_grid(CvSVM::P),
            CvParamGrid nu_grid = get_default_grid(CvSVM::NU),
            CvParamGrid coef_grid = get_default_grid(CvSVM::COEF),
            CvParamGrid degree_grid = get_default_grid(CvSVM::DEGREE) );
```

`k_fold` Cross-validation parameter. The training set is divided into `k_fold` subsets, one subset being used to train the model, the others forming the test set. So, the SVM algorithm is executed `k_fold` times.

The method trains the SVM model automatically by choosing the optimal parameters C, gamma, p, nu, coef0, degree from CvSVMParams . By optimal one means that the cross-validation estimate of the test set error is minimal. The parameters are iterated by a logarithmic grid, for example, the parameter gamma takes the values in the set (min, $min * step$, $min * step^2$, ... $min * step^n$) where min is gamma_grid.min_val, $step$ is gamma_grid.step, and n is the maximal index such, that

$$\texttt{gamma\_grid.min\_val} * \texttt{gamma\_grid.step}^n < \texttt{gamma\_grid.max\_val}$$

So step must always be greater than 1.

If there is no need in optimization in some parameter, the according grid step should be set to any value less or equal to 1. For example, to avoid optimization in gamma one should set gamma_grid.step = 0, gamma_grid.min_val, gamma_grid.max_val being arbitrary numbers. In this case, the value params.gamma will be taken for gamma.

And, finally, if the optimization in some parameter is required, but there is no idea of the corresponding grid, one may call the function CvSVM::get_default_grid. In order to generate a grid, say, for gamma, call CvSVM::get_default_grid(CvSVM::GAMMA).

This function works for the case of classification (params.svm_type=CvSVM::C_SVC or params.svm_type= as well as for the regression (params.svm_type=CvSVM::EPS_SVR or params.svm_type=CvSVM::NU_SVR). If params.svm_type=CvSVM::ONE_CLASS, no optimization is made and the usual SVM with specified in params parameters is executed.

CvSVM::get_default_grid

Generates a grid for the SVM parameters.

```
CvParamGrid CvSVM::get_default_grid( int param_id );
```

param_id Must be one of the following:

- **CvSVM::C**
- **CvSVM::GAMMA**
- **CvSVM::P**
- **CvSVM::NU**
- **CvSVM::COEF**
- **CvSVM::DEGREE** .

The grid will be generated for the parameter with this ID.

The function generates a grid for the specified parameter of the SVM algorithm. The grid may be passed to the function `CvSVM::train_auto`.

CvSVM::get_params

Returns the current SVM parameters.

```
CvSVMParams CvSVM::get_params() const;
```

This function may be used to get the optimal parameters that were obtained while automatically training `CvSVM::train_auto`.

CvSVM::get_support_vector*

Retrieves the number of support vectors and the particular vector.

```
int CvSVM::get_support_vector_count() const;
const float* CvSVM::get_support_vector(int i) const;
```

The methods can be used to retrieve the set of support vectors.

11.5 Decision Trees

The ML classes discussed in this section implement Classification And Regression Tree algorithms, which are described in [Breiman84].

The class CvDTree represents a single decision tree that may be used alone, or as a base class in tree ensembles (see Boosting and Random Trees).

A decision tree is a binary tree (i.e. tree where each non-leaf node has exactly 2 child nodes). It can be used either for classification, when each tree leaf is marked with some class label (multiple leafs may have the same label), or for regression, when each tree leaf is also assigned a constant (so the approximation function is piecewise constant).

Predicting with Decision Trees

To reach a leaf node, and to obtain a response for the input feature vector, the prediction procedure starts with the root node. From each non-leaf node the procedure goes to the left (i.e. selects the

left child node as the next observed node), or to the right based on the value of a certain variable, whose index is stored in the observed node. The variable can be either ordered or categorical. In the first case, the variable value is compared with the certain threshold (which is also stored in the node); if the value is less than the threshold, the procedure goes to the left, otherwise, to the right (for example, if the weight is less than 1 kilogram, the procedure goes to the left, else to the right). And in the second case the discrete variable value is tested to see if it belongs to a certain subset of values (also stored in the node) from a limited set of values the variable could take; if yes, the procedure goes to the left, else - to the right (for example, if the color is green or red, go to the left, else to the right). That is, in each node, a pair of entities (variable_index, decision_rule (threshold/subset)) is used. This pair is called a split (split on the variable variable_index). Once a leaf node is reached, the value assigned to this node is used as the output of prediction procedure.

Sometimes, certain features of the input vector are missed (for example, in the darkness it is difficult to determine the object color), and the prediction procedure may get stuck in the certain node (in the mentioned example if the node is split by color). To avoid such situations, decision trees use so-called surrogate splits. That is, in addition to the best ”primary” split, every tree node may also be split on one or more other variables with nearly the same results.

Training Decision Trees

The tree is built recursively, starting from the root node. All of the training data (feature vectors and the responses) is used to split the root node. In each node the optimum decision rule (i.e. the best ”primary” split) is found based on some criteria (in ML `gini` ”purity” criteria is used for classification, and sum of squared errors is used for regression). Then, if necessary, the surrogate splits are found that resemble the results of the primary split on the training data; all of the data is divided using the primary and the surrogate splits (just like it is done in the prediction procedure) between the left and the right child node. Then the procedure recursively splits both left and right nodes. At each node the recursive procedure may stop (i.e. stop splitting the node further) in one of the following cases:

- depth of the tree branch being constructed has reached the specified maximum value.
- number of training samples in the node is less than the specified threshold, when it is not statistically representative to split the node further.
- all the samples in the node belong to the same class (or, in the case of regression, the variation is too small).
- the best split found does not give any noticeable improvement compared to a random choice.

When the tree is built, it may be pruned using a cross-validation procedure, if necessary. That is, some branches of the tree that may lead to the model overfitting are cut off. Normally this

procedure is only applied to standalone decision trees, while tree ensembles usually build small enough trees and use their own protection schemes against overfitting.

Variable importance

Besides the obvious use of decision trees - prediction, the tree can be also used for various data analysis. One of the key properties of the constructed decision tree algorithms is that it is possible to compute importance (relative decisive power) of each variable. For example, in a spam filter that uses a set of words occurred in the message as a feature vector, the variable importance rating can be used to determine the most ”spam-indicating” words and thus help to keep the dictionary size reasonable.

Importance of each variable is computed over all the splits on this variable in the tree, primary and surrogate ones. Thus, to compute variable importance correctly, the surrogate splits must be enabled in the training parameters, even if there is no missing data.

[Breiman84] Breiman, L., Friedman, J. Olshen, R. and Stone, C. (1984), ”Classification and Regression Trees”, Wadsworth.

CvDTreeSplit

Decision tree node split.

```
struct CvDTreeSplit
{
    int var_idx;
    int inversed;
    float quality;
    CvDTreeSplit* next;
    union
    {
        int subset[2];
        struct
        {
            float c;
            int split_point;
        }
        ord;
    };
};
```

CvDTreeNode

Decision tree node.

```
struct CvDTreeNode
{
    int class_idx;
    int Tn;
    double value;

    CvDTreeNode* parent;
    CvDTreeNode* left;
    CvDTreeNode* right;

    CvDTreeSplit* split;

    int sample_count;
    int depth;
    ...
};
```

Other numerous fields of `CvDTreeNode` are used internally at the training stage.

CvDTreeParams

Decision tree training parameters.

```
struct CvDTreeParams
{
    int max_categories;
    int max_depth;
    int min_sample_count;
    int cv_folds;
    bool use_surrogates;
    bool use_1se_rule;
    bool truncate_pruned_tree;
    float regression_accuracy;
    const float* priors;

    CvDTreeParams() : max_categories(10), max_depth(INT_MAX), min_sample_count(10),
        cv_folds(10), use_surrogates(true), use_1se_rule(true),
        truncate_pruned_tree(true), regression_accuracy(0.01f), priors(0)
    {}

    CvDTreeParams( int _max_depth, int _min_sample_count,
                   float _regression_accuracy, bool _use_surrogates,
                   int _max_categories, int _cv_folds,
                   bool _use_1se_rule, bool _truncate_pruned_tree,
                   const float* _priors );
```

```
};
```

The structure contains all the decision tree training parameters. There is a default constructor that initializes all the parameters with the default values tuned for standalone classification tree. Any of the parameters can be overridden then, or the structure may be fully initialized using the advanced variant of the constructor.

CvDTreeTrainData

Decision tree training data and shared data for tree ensembles.

```
struct CvDTreeTrainData
{
    CvDTreeTrainData();
    CvDTreeTrainData( const CvMat* _train_data, int _tflag,
                      const CvMat* _responses, const CvMat* _var_idx=0,
                      const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                      const CvMat* _missing_mask=0,
                      const CvDTreeParams& _params=CvDTreeParams(),
                      bool _shared=false, bool _add_labels=false );
    virtual ~CvDTreeTrainData();

    virtual void set_data( const CvMat* _train_data, int _tflag,
                           const CvMat* _responses, const CvMat* _var_idx=0,
                           const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                           const CvMat* _missing_mask=0,
                           const CvDTreeParams& _params=CvDTreeParams(),
                           bool _shared=false, bool _add_labels=false,
                           bool _update_data=false );

    virtual void get_vectors( const CvMat* _subsample_idx,
         float* values, uchar* missing, float* responses,
         bool get_class_idx=false );

    virtual CvDTreeNode* subsample_data( const CvMat* _subsample_idx );

    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    // release all the data
    virtual void clear();

    int get_num_classes() const;
    int get_var_type(int vi) const;
    int get_work_var_count() const;
```

```
    virtual int* get_class_labels( CvDTreeNode* n );
    virtual float* get_ord_responses( CvDTreeNode* n );
    virtual int* get_labels( CvDTreeNode* n );
    virtual int* get_cat_var_data( CvDTreeNode* n, int vi );
    virtual CvPair32s32f* get_ord_var_data( CvDTreeNode* n, int vi );
    virtual int get_child_buf_idx( CvDTreeNode* n );

    ////////////////////////////////////

    virtual bool set_params( const CvDTreeParams& params );
    virtual CvDTreeNode* new_node( CvDTreeNode* parent, int count,
                                   int storage_idx, int offset );

    virtual CvDTreeSplit* new_split_ord( int vi, float cmp_val,
                int split_point, int inversed, float quality );
    virtual CvDTreeSplit* new_split_cat( int vi, float quality );
    virtual void free_node_data( CvDTreeNode* node );
    virtual void free_train_data();
    virtual void free_node( CvDTreeNode* node );

    int sample_count, var_all, var_count, max_c_count;
    int ord_var_count, cat_var_count;
    bool have_labels, have_priors;
    bool is_classifier;

    int buf_count, buf_size;
    bool shared;

    CvMat* cat_count;
    CvMat* cat_ofs;
    CvMat* cat_map;

    CvMat* counts;
    CvMat* buf;
    CvMat* direction;
    CvMat* split_buf;

    CvMat* var_idx;
    CvMat* var_type; // i-th element =
                     //   k<0  - ordered
                     //   k>=0 - categorical, see k-th element of cat_* arrays
    CvMat* priors;

    CvDTreeParams params;
```

```
    CvMemStorage* tree_storage;
    CvMemStorage* temp_storage;

    CvDTreeNode* data_root;

    CvSet* node_heap;
    CvSet* split_heap;
    CvSet* cv_heap;
    CvSet* nv_heap;

    CvRNG rng;
};
```

This structure is mostly used internally for storing both standalone trees and tree ensembles efficiently. Basically, it contains 3 types of information:

1. The training parameters, an instance of CvDTreeParams .
2. The training data, preprocessed in order to find the best splits more efficiently. For tree ensembles this preprocessed data is reused by all the trees. Additionally, the training data characteristics that are shared by all trees in the ensemble are stored here: variable types, the number of classes, class label compression map etc.
3. Buffers, memory storages for tree nodes, splits and other elements of the trees constructed.

There are 2 ways of using this structure. In simple cases (e.g. a standalone tree, or the ready-to-use "black box" tree ensemble from ML, like Random Trees or Boosting) there is no need to care or even to know about the structure - just construct the needed statistical model, train it and use it. The `CvDTreeTrainData` structure will be constructed and used internally. However, for custom tree algorithms, or another sophisticated cases, the structure may be constructed and used explicitly. The scheme is the following:

- The structure is initialized using the default constructor, followed by `set_data` (or it is built using the full form of constructor). The parameter `_shared` must be set to `true`.
- One or more trees are trained using this data, see the special form of the method `CvDTree::train`.
- Finally, the structure can be released only after all the trees using it are released.

CvDTree

Decision tree.

```
class CvDTree : public CvStatModel
{
public:
    CvDTree();
    virtual ~CvDTree();

    virtual bool train( const CvMat* _train_data, int _tflag,
                        const CvMat* _responses, const CvMat* _var_idx=0,
                        const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                        const CvMat* _missing_mask=0,
                        CvDTreeParams params=CvDTreeParams() );

    virtual bool train( CvDTreeTrainData* _train_data,
                        const CvMat* _subsample_idx );

    virtual CvDTreeNode* predict( const CvMat* _sample,
                                  const CvMat* _missing_data_mask=0,
                                  bool raw_mode=false ) const;
    virtual const CvMat* get_var_importance();
    virtual void clear();

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name );

    // special read & write methods for trees in the tree ensembles
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvDTreeTrainData* data );
    virtual void write( CvFileStorage* fs );

    const CvDTreeNode* get_root() const;
    int get_pruned_tree_idx() const;
    CvDTreeTrainData* get_data();

protected:

    virtual bool do_train( const CvMat* _subsample_idx );

    virtual void try_split_node( CvDTreeNode* n );
    virtual void split_node_data( CvDTreeNode* n );
    virtual CvDTreeSplit* find_best_split( CvDTreeNode* n );
    virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_surrogate_split_ord( CvDTreeNode* n, int vi );
```

```
    virtual CvDTreeSplit* find_surrogate_split_cat( CvDTreeNode* n, int vi );
    virtual double calc_node_dir( CvDTreeNode* node );
    virtual void complete_node_dir( CvDTreeNode* node );
    virtual void cluster_categories( const int* vectors, int vector_count,
        int var_count, int* sums, int k, int* cluster_labels );

    virtual void calc_node_value( CvDTreeNode* node );

    virtual void prune_cv();
    virtual double update_tree_rnc( int T, int fold );
    virtual int cut_tree( int T, int fold, double min_alpha );
    virtual void free_prune_data(bool cut_tree);
    virtual void free_tree();

    virtual void write_node( CvFileStorage* fs, CvDTreeNode* node );
    virtual void write_split( CvFileStorage* fs, CvDTreeSplit* split );
    virtual CvDTreeNode* read_node( CvFileStorage* fs,
                                    CvFileNode* node,
                                    CvDTreeNode* parent );
    virtual CvDTreeSplit* read_split( CvFileStorage* fs, CvFileNode* node );
    virtual void write_tree_nodes( CvFileStorage* fs );
    virtual void read_tree_nodes( CvFileStorage* fs, CvFileNode* node );

    CvDTreeNode* root;

    int pruned_tree_idx;
    CvMat* var_importance;

    CvDTreeTrainData* data;
};
```

CvDTree::train

Trains a decision tree.

```
bool CvDTree::train(
            const CvMat* _train_data,
            int _tflag,
            const CvMat* _responses,
            const CvMat* _var_idx=0,
            const CvMat* _sample_idx=0,
            const CvMat* _var_type=0,
```

```
            const CvMat* _missing_mask=0,
            CvDTreeParams params=CvDTreeParams() );

bool CvDTree::train( CvDTreeTrainData* _train_data, const CvMat*
_subsample_idx );
```

There are 2 `train` methods in `CvDTree`.

The first method follows the generic `CvStatModel::train` conventions, it is the most complete form. Both data layouts (`_tflag=CV_ROW_SAMPLE` and `_tflag=CV_COL_SAMPLE`) are supported, as well as sample and variable subsets, missing measurements, arbitrary combinations of input and output variable types etc. The last parameter contains all of the necessary training parameters, see the CvDTreeParams description.

The second method `train` is mostly used for building tree ensembles. It takes the pre-constructed CvDTreeTrainData instance and the optional subset of training set. The indices in `_subsample_idx` are counted relatively to the `_sample_idx`, passed to `CvDTreeTrainData` constructor. For example, if `_sample_idx=[1, 5, 7, 100]`, then `_subsample_idx=[0,3]` means that the samples `[1, 100]` of the original training set are used.

CvDTree::predict

Returns the leaf node of the decision tree corresponding to the input vector.

```
CvDTreeNode* CvDTree::predict(
            const CvMat* _sample,
            const CvMat* _missing_data_mask=0,
            bool raw_mode=false ) const;
```

The method takes the feature vector and the optional missing measurement mask on input, traverses the decision tree and returns the reached leaf node on output. The prediction result, either the class label or the estimated function value, may be retrieved as the `value` field of the CvDTreeNode structure, for example: dtree->predict(sample,mask)->value.

The last parameter is normally set to `false`, implying a regular input. If it is `true`, the method assumes that all the values of the discrete input variables have been already normalized to 0 to $num\_of\_categories_i - 1$ ranges. (as the decision tree uses such normalized representation internally). It is useful for faster prediction with tree ensembles. For ordered input variables the flag is not used.

Example: Building A Tree for Classifying Mushrooms. See the `mushroom.cpp` sample that demonstrates how to build and use the decision tree.

11.6 Boosting

A common machine learning task is supervised learning. In supervised learning, the goal is to learn the functional relationship $F : y = F(x)$ between the input x and the output y. Predicting the qualitative output is called classification, while predicting the quantitative output is called regression.

Boosting is a powerful learning concept, which provide a solution to the supervised classification learning task. It combines the performance of many "weak" classifiers to produce a powerful 'committee' HTF01 . A weak classifier is only required to be better than chance, and thus can be very simple and computationally inexpensive. Many of them smartly combined, however, results in a strong classifier, which often outperforms most 'monolithic' strong classifiers such as SVMs and Neural Networks.

Decision trees are the most popular weak classifiers used in boosting schemes. Often the simplest decision trees with only a single split node per tree (called stumps) are sufficient.

The boosted model is based on N training examples $(x_i, y_i)1N$ with $x_i \in R^K$ and $y_i \in -1, +1$. x_i is a K-component vector. Each component encodes a feature relevant for the learning task at hand. The desired two-class output is encoded as -1 and +1.

Different variants of boosting are known such as Discrete Adaboost, Real AdaBoost, LogitBoost, and Gentle AdaBoost FHT98 . All of them are very similar in their overall structure. Therefore, we will look only at the standard two-class Discrete AdaBoost algorithm as shown in the box below. Each sample is initially assigned the same weight (step 2). Next a weak classifier $f_{m(x)}$ is trained on the weighted training data (step 3a). Its weighted training error and scaling factor c_m is computed (step 3b). The weights are increased for training samples, which have been misclassified (step 3c). All weights are then normalized, and the process of finding the next weak classifier continues for another M-1 times. The final classifier $F(x)$ is the sign of the weighted sum over the individual weak classifiers (step 4).

- Given N examples $(x_i, y_i)1N$ with $x_i \in R^K, y_i \in -1, +1$.
- Start with weights $w_i = 1/N, i = 1, ..., N$.
- Repeat for m = $1, 2, ..., M$:
  - Fit the classifier $f_m(x) \in -1, 1$, using weights w_i on the training data.
  - Compute $err_m = E_w[1_{(y \neq f_m(x))}], c_m = log((1 - err_m)/err_m)$.
  - Set $w_i \Leftarrow w_i exp[c_m 1_{(y_i \neq f_m(x_i))}], i = 1, 2, ..., N$, and renormalize so that $\Sigma i w_i = 1$.
  - Output the classifier sign$[\Sigma m = 1 M c_m f_m(x)]$.

Two-class Discrete AdaBoost Algorithm: Training (steps 1 to 3) and Evaluation (step 4)

NOTE: As well as the classical boosting methods, the current implementation supports 2-class classifiers only. For M>2 classes there is the **AdaBoost.MH** algorithm, described in FHT98 , that reduces the problem to the 2-class problem, yet with a much larger training set.

In order to reduce computation time for boosted models without substantially losing accuracy, the influence trimming technique may be employed. As the training algorithm proceeds and the number of trees in the ensemble is increased, a larger number of the training samples are classified correctly and with increasing confidence, thereby those samples receive smaller weights on the subsequent iterations. Examples with very low relative weight have small impact on training of the weak classifier. Thus such examples may be excluded during the weak classifier training without having much effect on the induced classifier. This process is controlled with the weight_trim_rate parameter. Only examples with the summary fraction weight_trim_rate of the total weight mass are used in the weak classifier training. Note that the weights for **all** training examples are recomputed at each training iteration. Examples deleted at a particular iteration may be used again for learning some of the weak classifiers further FHT98 .

[HTF01] Hastie, T., Tibshirani, R., Friedman, J. H. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. 2001.

[FHT98] Friedman, J. H., Hastie, T. and Tibshirani, R. Additive Logistic Regression: a Statistical View of Boosting. Technical Report, Dept. of Statistics, Stanford University, 1998.

CvBoostParams

Boosting training parameters.

```
struct CvBoostParams : public CvDTreeParams
{
    int boost_type;
    int weak_count;
    int split_criteria;
    double weight_trim_rate;

    CvBoostParams();
    CvBoostParams( int boost_type, int weak_count, double weight_trim_rate,
                   int max_depth, bool use_surrogates, const float* priors );
};
```

The structure is derived from CvDTreeParams , but not all of the decision tree parameters are supported. In particular, cross-validation is not supported.

CvBoostTree

Weak tree classifier.

```
class CvBoostTree: public CvDTree
{
public:
    CvBoostTree();
    virtual ~CvBoostTree();

    virtual bool train( CvDTreeTrainData* _train_data,
                        const CvMat* subsample_idx, CvBoost* ensemble );
    virtual void scale( double s );
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvBoost* ensemble, CvDTreeTrainData* _data );
    virtual void clear();

protected:
    ...
    CvBoost* ensemble;
};
```

The weak classifier, a component of the boosted tree classifier CvBoost , is a derivative of CvDTree . Normally, there is no need to use the weak classifiers directly, however they can be accessed as elements of the sequence `CvBoost::weak`, retrieved by `CvBoost::get_weak_predictors`.

Note, that in the case of LogitBoost and Gentle AdaBoost each weak predictor is a regression tree, rather than a classification tree. Even in the case of Discrete AdaBoost and Real AdaBoost the `CvBoostTree::predict` return value (`CvDTreeNode::value`) is not the output class label; a negative value "votes" for class #0, a positive - for class #1. And the votes are weighted. The weight of each individual tree may be increased or decreased using the method `CvBoostTree::scale`.

CvBoost

Boosted tree classifier.

```
class CvBoost : public CvStatModel
{
public:
    // Boosting type
    enum { DISCRETE=0, REAL=1, LOGIT=2, GENTLE=3 };

    // Splitting criteria
    enum { DEFAULT=0, GINI=1, MISCLASS=3, SQERR=4 };

    CvBoost();
    virtual ~CvBoost();
```

```
    CvBoost( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams() );

    virtual bool train( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams(),
             bool update=false );

    virtual float predict( const CvMat* _sample, const CvMat* _missing=0,
                           CvMat* weak_responses=0, CvSlice slice=CV_WHOLE_SEQ,
                           bool raw_mode=false ) const;

    virtual void prune( CvSlice slice );

    virtual void clear();

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );

    CvSeq* get_weak_predictors();
    const CvBoostParams& get_params() const;
    ...

protected:
    virtual bool set_params( const CvBoostParams& _params );
    virtual void update_weights( CvBoostTree* tree );
    virtual void trim_weights();
    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvDTreeTrainData* data;
    CvBoostParams params;
    CvSeq* weak;
    ...
};
```

CvBoost::train

Trains a boosted tree classifier.

```
bool CvBoost::train(
           const CvMat* _train_data,
           int _tflag,
           const CvMat* _responses,
           const CvMat* _var_idx=0,
           const CvMat* _sample_idx=0,
           const CvMat* _var_type=0,
           const CvMat* _missing_mask=0,
           CvBoostParams params=CvBoostParams(),
           bool update=false );
```

The train method follows the common template; the last parameter `update` specifies whether the classifier needs to be updated (i.e. the new weak tree classifiers added to the existing ensemble), or the classifier needs to be rebuilt from scratch. The responses must be categorical, i.e. boosted trees can not be built for regression, and there should be 2 classes.

CvBoost::predict

Predicts a response for the input sample.

```
float CvBoost::predict(
           const CvMat* sample,
           const CvMat* missing=0,
           CvMat* weak_responses=0,
           CvSlice slice=CV_WHOLE_SEQ,
           bool raw_mode=false ) const;
```

The method `CvBoost::predict` runs the sample through the trees in the ensemble and returns the output class label based on the weighted voting.

CvBoost::prune

Removes the specified weak classifiers.

```
void CvBoost::prune( CvSlice slice );
```

The method removes the specified weak classifiers from the sequence. Note that this method should not be confused with the pruning of individual decision trees, which is currently not supported.

CvBoost::get_weak_predictors

Returns the sequence of weak tree classifiers.

```
CvSeq* CvBoost::get_weak_predictors();
```

The method returns the sequence of weak classifiers. Each element of the sequence is a pointer to a `CvBoostTree` class (or, probably, to some of its derivatives).

11.7 Random Trees

Random trees have been introduced by Leo Breiman and Adele Cutler: http://www.stat.berkeley.edu/users/breiman/RandomForests/. The algorithm can deal with both classification and regression problems. Random trees is a collection (ensemble) of tree predictors that is called **forest** further in this section (the term has been also introduced by L. Breiman). The classification works as follows: the random trees classifier takes the input feature vector, classifies it with every tree in the forest, and outputs the class label that recieved the majority of "votes". In the case of regression the classifier response is the average of the responses over all the trees in the forest.

All the trees are trained with the same parameters, but on the different training sets, which are generated from the original training set using the bootstrap procedure: for each training set we randomly select the same number of vectors as in the original set (=N). The vectors are chosen with replacement. That is, some vectors will occur more than once and some will be absent. At each node of each tree trained not all the variables are used to find the best split, rather than a random subset of them. With each node a new subset is generated, however its size is fixed for all the nodes and all the trees. It is a training parameter, set to $\sqrt{number\_of\_variables}$ by default. None of the trees that are built are pruned.

In random trees there is no need for any accuracy estimation procedures, such as cross-validation or bootstrap, or a separate test set to get an estimate of the training error. The error is estimated internally during the training. When the training set for the current tree is drawn by

sampling with replacement, some vectors are left out (so-called *oob (out-of-bag) data*). The size of oob data is about `N/3`. The classification error is estimated by using this oob-data as following:

- Get a prediction for each vector, which is oob relatively to the i-th tree, using the very i-th tree.
- After all the trees have been trained, for each vector that has ever been oob, find the class-"winner" for it (i.e. the class that has got the majority of votes in the trees, where the vector was oob) and compare it to the ground-truth response.
- Then the classification error estimate is computed as ratio of number of misclassified oob vectors to all the vectors in the original data. In the case of regression the oob-error is computed as the squared error for oob vectors difference divided by the total number of vectors.

References:

- Machine Learning, Wald I, July 2002. http://stat-www.berkeley.edu/users/breiman/wald2002-1.pdf
- Looking Inside the Black Box, Wald II, July 2002. http://stat-www.berkeley.edu/users/breiman/wald2002-2.pdf
- Software for the Masses, Wald III, July 2002. http://stat-www.berkeley.edu/users/breiman/wald2002-3.pdf
- And other articles from the web site http://www.stat.berkeley.edu/users/breiman/RandomForests/cc_home.htm.

CvRTParams

Training Parameters of Random Trees.

```
struct CvRTParams : public CvDTreeParams
{
    bool calc_var_importance;
    int nactive_vars;
    CvTermCriteria term_crit;

    CvRTParams() : CvDTreeParams( 5, 10, 0, false, 10, 0, false, false, 0 ),
        calc_var_importance(false), nactive_vars(0)
    {
        term_crit = cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 50, 0.1 );
    }
```

```
    CvRTParams( int _max_depth, int _min_sample_count,
                float _regression_accuracy, bool _use_surrogates,
                int _max_categories, const float* _priors,
                bool _calc_var_importance,
                int _nactive_vars, int max_tree_count,
                float forest_accuracy, int termcrit_type );
};
```

The set of training parameters for the forest is the superset of the training parameters for a single tree. However, Random trees do not need all the functionality/features of decision trees, most noticeably, the trees are not pruned, so the cross-validation parameters are not used.

CvRTrees

Random Trees.

```
class CvRTrees : public CvStatModel
{
public:
    CvRTrees();
    virtual ~CvRTrees();
    virtual bool train( const CvMat* _train_data, int _tflag,
                        const CvMat* _responses, const CvMat* _var_idx=0,
                        const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                        const CvMat* _missing_mask=0,
                        CvRTParams params=CvRTParams() );
    virtual float predict( const CvMat* sample, const CvMat* missing = 0 )
                                                                    const;
    virtual void clear();

    virtual const CvMat* get_var_importance();
    virtual float get_proximity( const CvMat* sample_1, const CvMat* sample_2 )
                                                                      const;

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name );

    CvMat* get_active_var_mask();
    CvRNG* get_rng();

    int get_tree_count() const;
    CvForestTree* get_tree(int i) const;

protected:
```

```
    bool grow_forest( const CvTermCriteria term_crit );

    // array of the trees of the forest
    CvForestTree** trees;
    CvDTreeTrainData* data;
    int ntrees;
    int nclasses;
    ...
};
```

CvRTrees::train

Trains the Random Trees model.

```
bool CvRTrees::train(
           const CvMat* train_data,
           int tflag,
           const CvMat* responses,
           const CvMat* comp_idx=0,
           const CvMat* sample_idx=0,
           const CvMat* var_type=0,
           const CvMat* missing_mask=0,
           CvRTParams params=CvRTParams() );
```

The method `CvRTrees::train` is very similar to the first form of `CvDTree::train`() and follows the generic method `CvStatModel::train` conventions. All of the specific to the algorithm training parameters are passed as a CvRTParams instance. The estimate of the training error (`oob-error`) is stored in the protected class member `oob_error`.

CvRTrees::predict

Predicts the output for the input sample.

```
double CvRTrees::predict(
           const CvMat* sample,
           const CvMat* missing=0 ) const;
```

The input parameters of the prediction method are the same as in `CvDTree::predict`, but the return value type is different. This method returns the cumulative result from all the trees in the forest (the class that receives the majority of voices, or the mean of the regression function estimates).

CvRTrees::get_var_importance

Retrieves the variable importance array.

```
const CvMat* CvRTrees::get_var_importance() const;
```

The method returns the variable importance vector, computed at the training stage when `CvRTParams ::calc_var_importance` is set. If the training flag is not set, then the `NULL` pointer is returned. This is unlike decision trees, where variable importance can be computed anytime after the training.

CvRTrees::get_proximity

Retrieves the proximity measure between two training samples.

```
float CvRTrees::get_proximity(
            const CvMat* sample_1,
            const CvMat* sample_2 ) const;
```

The method returns proximity measure between any two samples (the ratio of the those trees in the ensemble, in which the samples fall into the same leaf node, to the total number of the trees).

Example: Prediction of mushroom goodness using random trees classifier

```
#include <float.h>
#include <stdio.h>
#include <ctype.h>
#include "ml.h"

int main( void )
{
    CvStatModel*    cls = NULL;
    CvFileStorage*  storage = cvOpenFileStorage( "Mushroom.xml",
```

```
                                                    NULL,CV_STORAGE_READ );
CvMat*          data = (CvMat*)cvReadByName(storage, NULL, "sample", 0 );
CvMat           train_data, test_data;
CvMat           response;
CvMat*          missed = NULL;
CvMat*          comp_idx = NULL;
CvMat*          sample_idx = NULL;
CvMat*          type_mask = NULL;
int             resp_col = 0;
int             i,j;
CvRTreesParams  params;
CvTreeClassifierTrainParams cart_params;
const int       ntrain_samples = 1000;
const int       ntest_samples  = 1000;
const int       nvars = 23;

if(data == NULL || data->cols != nvars)
{
    puts("Error in source data");
    return -1;
}

cvGetSubRect( data, &train_data, cvRect(0, 0, nvars, ntrain_samples) );
cvGetSubRect( data, &test_data, cvRect(0, ntrain_samples, nvars,
    ntrain_samples + ntest_samples) );

resp_col = 0;
cvGetCol( &train_data, &response, resp_col);

/* create missed variable matrix */
missed = cvCreateMat(train_data.rows, train_data.cols, CV_8UC1);
for( i = 0; i < train_data.rows; i++ )
    for( j = 0; j < train_data.cols; j++ )
        CV_MAT_ELEM(*missed,uchar,i,j)
      = (uchar)(CV_MAT_ELEM(train_data,float,i,j) < 0);

/* create comp_idx vector */
comp_idx = cvCreateMat(1, train_data.cols-1, CV_32SC1);
for( i = 0; i < train_data.cols; i++ )
{
    if(i<resp_col)CV_MAT_ELEM(*comp_idx,int,0,i) = i;
    if(i>resp_col)CV_MAT_ELEM(*comp_idx,int,0,i-1) = i;
}

/* create sample_idx vector */
```

```
    sample_idx = cvCreateMat(1, train_data.rows, CV_32SC1);
    for( j = i = 0; i < train_data.rows; i++ )
    {
        if(CV_MAT_ELEM(response,float,i,0) < 0) continue;
        CV_MAT_ELEM(*sample_idx,int,0,j) = i;
        j++;
    }
    sample_idx->cols = j;

    /* create type mask */
    type_mask = cvCreateMat(1, train_data.cols+1, CV_8UC1);
    cvSet( type_mask, cvRealScalar(CV_VAR_CATEGORICAL), 0);

    // initialize training parameters
    cvSetDefaultParamTreeClassifier((CvStatModelParams*)&cart_params);
    cart_params.wrong_feature_as_unknown = 1;
    params.tree_params = &cart_params;
    params.term_crit.max_iter = 50;
    params.term_crit.epsilon = 0.1;
    params.term_crit.type = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

    puts("Random forest results");
    cls = cvCreateRTreesClassifier( &train_data,
                                    CV_ROW_SAMPLE,
                                    &response,
                                    (CvStatModelParams*)&
                                    params,
                                    comp_idx,
                                    sample_idx,
                                    type_mask,
                                    missed );
    if( cls )
    {
        CvMat sample = cvMat( 1, nvars, CV_32FC1, test_data.data.fl );
        CvMat test_resp;
        int wrong = 0, total = 0;
        cvGetCol( &test_data, &test_resp, resp_col);
        for( i = 0; i < ntest_samples; i++, sample.data.fl += nvars )
        {
            if( CV_MAT_ELEM(test_resp,float,i,0) >= 0 )
            {
                float resp = cls->predict( cls, &sample, NULL );
                wrong += (fabs(resp-response.data.fl[i]) > 1e-3 ) ? 1 : 0;
                total++;
            }
```

```
        }
        printf( "Test set error = %.2f\n", wrong*100.f/(float)total );
    }
    else
       puts("Error forest creation");

    cvReleaseMat(&missed);
    cvReleaseMat(&sample_idx);
    cvReleaseMat(&comp_idx);
    cvReleaseMat(&type_mask);
    cvReleaseMat(&data);
    cvReleaseStatModel(&cls);
    cvReleaseFileStorage(&storage);
    return 0;
}
```

11.8 Expectation-Maximization

The EM (Expectation-Maximization) algorithm estimates the parameters of the multivariate probability density function in the form of a Gaussian mixture distribution with a specified number of mixtures.

Consider the set of the feature vectors $x_1, x_2, ..., x_N$: N vectors from a d-dimensional Euclidean space drawn from a Gaussian mixture:

$$p(x; a_k, S_k, \pi_k) = \sum_{k=1}^{m} \pi_k p_k(x), \quad \pi_k \geq 0, \quad \sum_{k=1}^{m} \pi_k = 1,$$

$$p_k(x) = \varphi(x; a_k, S_k) = \frac{1}{(2\pi)^{d/2} \mid S_k \mid^{1/2}} exp\left\{-\frac{1}{2}(x-a_k)^T S_k^{-1}(x-a_k)\right\},$$

where m is the number of mixtures, p_k is the normal distribution density with the mean a_k and covariance matrix S_k, π_k is the weight of the k-th mixture. Given the number of mixtures M and the samples x_i, $i = 1..N$ the algorithm finds the maximum-likelihood estimates (MLE) of the all the mixture parameters, i.e. a_k, S_k and π_k :

$$L(x, \theta) = logp(x, \theta) = \sum_{i=1}^{N} log\left(\sum_{k=1}^{m} \pi_k p_k(x)\right) \rightarrow \max_{\theta \in \Theta},$$

$$\Theta = \left\{(a_k, S_k, \pi_k) : a_k \in \mathbb{R}^d, S_k = S_k^T > 0, S_k \in \mathbb{R}^{d \times d}, \pi_k \geq 0, \sum_{k=1}^{m} \pi_k = 1\right\}.$$

EM algorithm is an iterative procedure. Each iteration of it includes two steps. At the first step (Expectation-step, or E-step), we find a probability $p_{i,k}$ (denoted $\alpha_{i,k}$ in the formula below) of sample `i` to belong to mixture `k` using the currently available mixture parameter estimates:

$$\alpha_{ki} = \frac{\pi_k \varphi(x; a_k, S_k)}{\sum_{j=1}^{m} \pi_j \varphi(x; a_j, S_j)}.$$

At the second step (Maximization-step, or M-step) the mixture parameter estimates are refined using the computed probabilities:

$$\pi_k = \frac{1}{N} \sum_{i=1}^{N} \alpha_{ki}, \quad a_k = \frac{\sum_{i=1}^{N} \alpha_{ki} x_i}{\sum_{i=1}^{N} \alpha_{ki}}, \quad S_k = \frac{\sum_{i=1}^{N} \alpha_{ki} (x_i - a_k)(x_i - a_k)^T}{\sum_{i=1}^{N} \alpha_{ki}},$$

Alternatively, the algorithm may start with the M-step when the initial values for $p_{i,k}$ can be provided. Another alternative when $p_{i,k}$ are unknown, is to use a simpler clustering algorithm to pre-cluster the input samples and thus obtain initial $p_{i,k}$. Often (and in ML) the KMeans2 algorithm is used for that purpose.

One of the main that EM algorithm should deal with is the large number of parameters to estimate. The majority of the parameters sits in covariance matrices, which are $d \times d$ elements each (where d is the feature space dimensionality). However, in many practical problems the covariance matrices are close to diagonal, or even to $\mu_k * I$, where I is identity matrix and μ_k is mixture-dependent "scale" parameter. So a robust computation scheme could be to start with the harder constraints on the covariance matrices and then use the estimated parameters as an input for a less constrained optimization problem (often a diagonal covariance matrix is already a good enough approximation).

References:

- Bilmes98 J. A. Bilmes. A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models. Technical Report TR-97-021, International Computer Science Institute and Computer Science Division, University of California at Berkeley, April 1998.

CvEMParams

Parameters of the EM algorithm.

```
struct CvEMParams
{
    CvEMParams() : nclusters(10), cov_mat_type(CvEM::COV_MAT_DIAGONAL),
```

```
        start_step(CvEM::START_AUTO_STEP), probs(0), weights(0), means(0),
                                                    covs(0)
    {
        term_crit=cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
                                                100, FLT_EPSILON );
    }

    CvEMParams( int _nclusters, int _cov_mat_type=1/*CvEM::COV_MAT_DIAGONAL*/,
                int _start_step=0/*CvEM::START_AUTO_STEP*/,
                CvTermCriteria _term_crit=cvTermCriteria(
                                    CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
                                    100, FLT_EPSILON),
                CvMat* _probs=0, CvMat* _weights=0,
                CvMat* _means=0, CvMat** _covs=0 ) :
                nclusters(_nclusters), cov_mat_type(_cov_mat_type),
                start_step(_start_step),
                probs(_probs), weights(_weights), means(_means), covs(_covs),
                term_crit(_term_crit)
    {}

    int nclusters;
    int cov_mat_type;
    int start_step;
    const CvMat* probs;
    const CvMat* weights;
    const CvMat* means;
    const CvMat** covs;
    CvTermCriteria term_crit;
};
```

The structure has 2 constructors, the default one represents a rough rule-of-thumb, with another one it is possible to override a variety of parameters, from a single number of mixtures (the only essential problem-dependent parameter), to the initial values for the mixture parameters.

CvEM

EM model.

```
class CV_EXPORTS CvEM : public CvStatModel
{
public:
    // Type of covariance matrices
    enum { COV_MAT_SPHERICAL=0, COV_MAT_DIAGONAL=1, COV_MAT_GENERIC=2 };

    // The initial step
```

```
    enum { START_E_STEP=1, START_M_STEP=2, START_AUTO_STEP=0 };

    CvEM();
    CvEM( const CvMat* samples, const CvMat* sample_idx=0,
          CvEMParams params=CvEMParams(), CvMat* labels=0 );
    virtual ~CvEM();

    virtual bool train( const CvMat* samples, const CvMat* sample_idx=0,
                        CvEMParams params=CvEMParams(), CvMat* labels=0 );

    virtual float predict( const CvMat* sample, CvMat* probs ) const;
    virtual void clear();

    int get_nclusters() const { return params.nclusters; }
    const CvMat* get_means() const { return means; }
    const CvMat** get_covs() const { return covs; }
    const CvMat* get_weights() const { return weights; }
    const CvMat* get_probs() const { return probs; }

protected:

    virtual void set_params( const CvEMParams& params,
                             const CvVectors& train_data );
    virtual void init_em( const CvVectors& train_data );
    virtual double run_em( const CvVectors& train_data );
    virtual void init_auto( const CvVectors& samples );
    virtual void kmeans( const CvVectors& train_data, int nclusters,
                         CvMat* labels, CvTermCriteria criteria,
                         const CvMat* means );
    CvEMParams params;
    double log_likelihood;

    CvMat* means;
    CvMat** covs;
    CvMat* weights;
    CvMat* probs;

    CvMat* log_weight_div_det;
    CvMat* inv_eigen_values;
    CvMat** cov_rotate_mats;
};
```

CvEM::train

Estimates the Gaussian mixture parameters from the sample set.

```
void CvEM::train(
            const CvMat* samples,
            const CvMat* sample_idx=0,
            CvEMParams params=CvEMParams(),
            CvMat* labels=0 );
```

Unlike many of the ML models, EM is an unsupervised learning algorithm and it does not take responses (class labels or the function values) on input. Instead, it computes the MLE of the Gaussian mixture parameters from the input sample set, stores all the parameters inside the structure: $p_{i,k}$ in `probs`, a_k in `means` S_k in `covs[k]`, π_k in `weights` and optionally computes the output "class label" for each sample: $\texttt{labels}_i = \texttt{arg max}_k(p_{i,k}), i = 1..N$ (i.e. indices of the most-probable mixture for each sample).

The trained model can be used further for prediction, just like any other classifier. The model trained is similar to the Bayes classifier .

Example: Clustering random samples of multi-Gaussian distribution using EM

```
#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    const int N = 4;
    const int N1 = (int)sqrt((double)N);
    const CvScalar colors[] = {{0,0,255}},{{0,255,0}},
                                    {{0,255,255}},{{255,255,0}
                                    ;
    int i, j;
    int nsamples = 100;
    CvRNG rng_state = cvRNG(-1);
    CvMat* samples = cvCreateMat( nsamples, 2, CV_32FC1 );
    CvMat* labels = cvCreateMat( nsamples, 1, CV_32SC1 );
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    float _sample[2];
    CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
    CvEM em_model;
    CvEMParams params;
    CvMat samples_part;
```

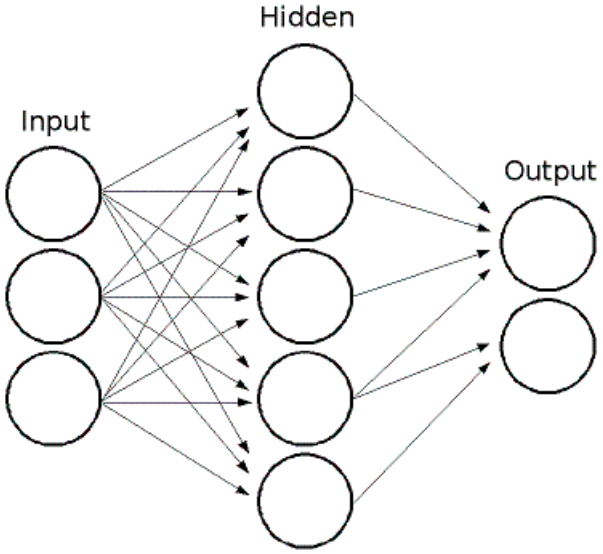
```
    cvReshape( samples, samples, 2, 0 );
    for( i = 0; i < N; i++ )
    {
        CvScalar mean, sigma;

        // form the training samples
        cvGetRows( samples, &samples_part, i*nsamples/N,
                                           (i+1)*nsamples/N );
        mean = cvScalar(((i%N1)+1.)*img->width/(N1+1),
                        ((i/N1)+1.)*img->height/(N1+1));
        sigma = cvScalar(30,30);
        cvRandArr( &rng_state, &samples_part, CV_RAND_NORMAL,
                                                    mean, sigma );
    }
    cvReshape( samples, samples, 1, 0 );

    // initialize model's parameters
    params.covs      = NULL;
    params.means     = NULL;
    params.weights   = NULL;
    params.probs     = NULL;
    params.nclusters = N;
    params.cov_mat_type       = CvEM::COV_MAT_SPHERICAL;
    params.start_step         = CvEM::START_AUTO_STEP;
    params.term_crit.max_iter = 10;
    params.term_crit.epsilon  = 0.1;
    params.term_crit.type     = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

    // cluster the data
    em_model.train( samples, 0, params, labels );

#if 0
    // the piece of code shows how to repeatedly optimize the model
    // with less-constrained parameters
    //(COV_MAT_DIAGONAL instead of COV_MAT_SPHERICAL)
    // when the output of the first stage is used as input for the second.
    CvEM em_model2;
    params.cov_mat_type = CvEM::COV_MAT_DIAGONAL;
    params.start_step = CvEM::START_E_STEP;
    params.means = em_model.get_means();
    params.covs = (const CvMat**)em_model.get_covs();
    params.weights = em_model.get_weights();

    em_model2.train( samples, 0, params, labels );
```

```
    // to use em_model2, replace em_model.predict()
    // with em_model2.predict() below
#endif
    // classify every image pixel
    cvZero( img );
    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            CvPoint pt = cvPoint(j, i);
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;
            int response = cvRound(em_model.predict( &sample, NULL ));
            CvScalar c = colors[response];

            cvCircle( img, pt, 1, cvScalar(c.val[0]*0.75,
                c.val[1]*0.75,c.val[2]*0.75), CV_FILLED );
        }
    }

    //draw the clustered samples
    for( i = 0; i < nsamples; i++ )
    {
        CvPoint pt;
        pt.x = cvRound(samples->data.fl[i*2]);
        pt.y = cvRound(samples->data.fl[i*2+1]);
        cvCircle( img, pt, 1, colors[labels->data.i[i]], CV_FILLED );
    }

    cvNamedWindow( "EM-clustering result", 1 );
    cvShowImage( "EM-clustering result", img );
    cvWaitKey(0);

    cvReleaseMat( &samples );
    cvReleaseMat( &labels );
    return 0;
}
```

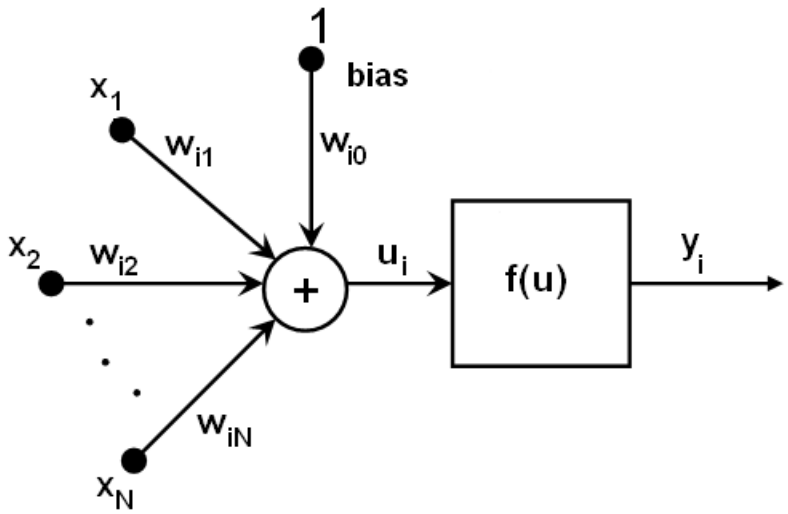
11.9 Neural Networks

ML implements feed-forward artificial neural networks, more particularly, multi-layer perceptrons (MLP), the most commonly used type of neural networks. MLP consists of the input layer, output layer and one or more hidden layers. Each layer of MLP includes one or more neurons that are

directionally linked with the neurons from the previous and the next layer. Here is an example of a 3-layer perceptron with 3 inputs, 2 outputs and the hidden layer including 5 neurons:



All the neurons in MLP are similar. Each of them has several input links (i.e. it takes the output values from several neurons in the previous layer on input) and several output links (i.e. it passes the response to several neurons in the next layer). The values retrieved from the previous layer are summed with certain weights, individual for each neuron, plus the bias term, and the sum is transformed using the activation function f that may be also different for different neurons. Here is the picture:



In other words, given the outputs x_j of the layer n, the outputs y_i of the layer $n+1$ are computed

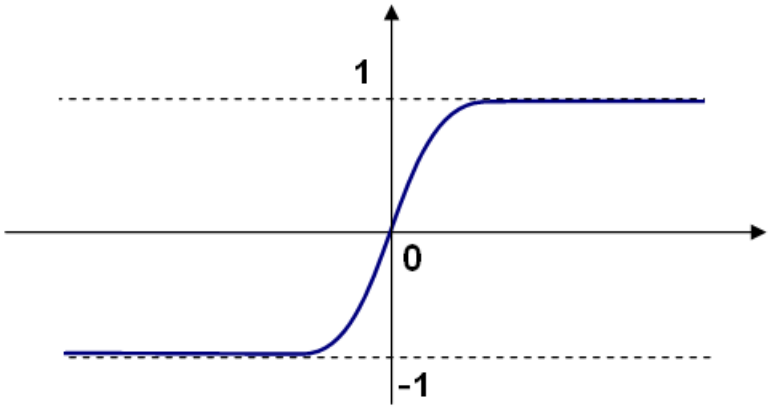
as:

$$u_i = \sum_j (w_{i,j}^{n+1} * x_j) + w_{i,bias}^{n+1}$$

$$y_i = f(u_i)$$

Different activation functions may be used, ML implements 3 standard ones:

- Identity function (`CvANN_MLP::IDENTITY`): $f(x) = x$
- Symmetrical sigmoid (`CvANN_MLP::SIGMOID_SYM`): $f(x) = \beta * (1 - e^{-\alpha x})/(1 + e^{-\alpha x})$, the default choice for MLP; the standard sigmoid with $\beta = 1, \alpha = 1$ is shown below:



- Gaussian function (`CvANN_MLP::GAUSSIAN`): $f(x) = \beta e^{-\alpha x * x}$, not completely supported by the moment.

In ML all the neurons have the same activation functions, with the same free parameters (α, β) that are specified by user and are not altered by the training algorithms.

So the whole trained network works as follows: It takes the feature vector on input, the vector size is equal to the size of the input layer, when the values are passed as input to the first hidden layer, the outputs of the hidden layer are computed using the weights and the activation functions and passed further downstream, until we compute the output layer.

So, in order to compute the network one needs to know all the weights $w_{i,j}^{n+1)}$. The weights are computed by the training algorithm. The algorithm takes a training set: multiple input vectors with the corresponding output vectors, and iteratively adjusts the weights to try to make the network give the desired response on the provided input vectors.

The larger the network size (the number of hidden layers and their sizes), the more is the potential network flexibility, and the error on the training set could be made arbitrarily small. But

at the same time the learned network will also ”learn” the noise present in the training set, so the error on the test set usually starts increasing after the network size reaches some limit. Besides, the larger networks are train much longer than the smaller ones, so it is reasonable to preprocess the data (using CalcPCA or similar technique) and train a smaller network on only the essential features.

Another feature of the MLP's is their inability to handle categorical data as is, however there is a workaround. If a certain feature in the input or output (i.e. in the case of `n`-class classifier for $n > 2$) layer is categorical and can take $M > 2$ different values, it makes sense to represent it as binary tuple of `M` elements, where `i`-th element is 1 if and only if the feature is equal to the `i`-th value out of `M` possible. It will increase the size of the input/output layer, but will speedup the training algorithm convergence and at the same time enable ”fuzzy” values of such variables, i.e. a tuple of probabilities instead of a fixed value.

ML implements 2 algorithms for training MLP's. The first is the classical random sequential back-propagation algorithm and the second (default one) is batch RPROP algorithm.

References:

- http://en.wikipedia.org/wiki/Backpropagation. Wikipedia article about the back-propagation algorithm.
- Y. LeCun, L. Bottou, G.B. Orr and K.-R. Muller, ”Efficient backprop”, in Neural Networks—Tricks of the Trade, Springer Lecture Notes in Computer Sciences 1524, pp.5-50, 1998.
- M. Riedmiller and H. Braun, ”A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm”, Proc. ICNN, San Francisco (1993).

CvANN_MLP_TrainParams

Parameters of the MLP training algorithm.

```
struct CvANN_MLP_TrainParams
{
    CvANN_MLP_TrainParams();
    CvANN_MLP_TrainParams( CvTermCriteria term_crit, int train_method,
                           double param1, double param2=0 );
    ~CvANN_MLP_TrainParams();

    enum { BACKPROP=0, RPROP=1 };

    CvTermCriteria term_crit;
    int train_method;

    // backpropagation parameters
    double bp_dw_scale, bp_moment_scale;
```

```
    // rprop parameters
    double rp_dw0, rp_dw_plus, rp_dw_minus, rp_dw_min, rp_dw_max;
};
```

The structure has default constructor that initializes parameters for RPROP algorithm. There is also more advanced constructor to customize the parameters and/or choose backpropagation algorithm. Finally, the individual parameters can be adjusted after the structure is created.

CvANN_MLP

MLP model.

```
class CvANN_MLP : public CvStatModel
{
public:
    CvANN_MLP();
    CvANN_MLP( const CvMat* _layer_sizes,
               int _activ_func=SIGMOID_SYM,
               double _f_param1=0, double _f_param2=0 );

    virtual ~CvANN_MLP();

    virtual void create( const CvMat* _layer_sizes,
                         int _activ_func=SIGMOID_SYM,
                         double _f_param1=0, double _f_param2=0 );

    virtual int train( const CvMat* _inputs, const CvMat* _outputs,
                       const CvMat* _sample_weights,
                       const CvMat* _sample_idx=0,
                       CvANN_MLP_TrainParams _params = CvANN_MLP_TrainParams(),
                       int flags=0 );
    virtual float predict( const CvMat* _inputs,
                           CvMat* _outputs ) const;

    virtual void clear();

    // possible activation functions
    enum { IDENTITY = 0, SIGMOID_SYM = 1, GAUSSIAN = 2 };

    // available training flags
    enum { UPDATE_WEIGHTS = 1, NO_INPUT_SCALE = 2, NO_OUTPUT_SCALE = 4 };

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* storage, const char* name );
```

```
    int get_layer_count() { return layer_sizes ? layer_sizes->cols : 0; }
    const CvMat* get_layer_sizes() { return layer_sizes; }

protected:

    virtual bool prepare_to_train( const CvMat* _inputs, const CvMat* _outputs,
            const CvMat* _sample_weights, const CvMat* _sample_idx,
            CvANN_MLP_TrainParams _params,
            CvVectors* _ivecs, CvVectors* _ovecs, double** _sw, int _flags );

    // sequential random backpropagation
    virtual int train_backprop( CvVectors _ivecs, CvVectors _ovecs,
                                            const double* _sw );

    // RPROP algorithm
    virtual int train_rprop( CvVectors _ivecs, CvVectors _ovecs,
                                          const double* _sw );

    virtual void calc_activ_func( CvMat* xf, const double* bias ) const;
    virtual void calc_activ_func_deriv( CvMat* xf, CvMat* deriv,
                                          const double* bias ) const;
    virtual void set_activ_func( int _activ_func=SIGMOID_SYM,
                                 double _f_param1=0, double _f_param2=0 );
    virtual void init_weights();
    virtual void scale_input( const CvMat* _src, CvMat* _dst ) const;
    virtual void scale_output( const CvMat* _src, CvMat* _dst ) const;
    virtual void calc_input_scale( const CvVectors* vecs, int flags );
    virtual void calc_output_scale( const CvVectors* vecs, int flags );

    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvMat* layer_sizes;
    CvMat* wbuf;
    CvMat* sample_weights;
    double** weights;
    double f_param1, f_param2;
    double min_val, max_val, min_val1, max_val1;
    int activ_func;
    int max_count, max_buf_sz;
    CvANN_MLP_TrainParams params;
    CvRNG rng;
};
```

Unlike many other models in ML that are constructed and trained at once, in the MLP model these steps are separated. First, a network with the specified topology is created using the non-default constructor or the method `create`. All the weights are set to zeros. Then the network is trained using the set of input and output vectors. The training procedure can be repeated more than once, i.e. the weights can be adjusted based on the new training data.

CvANN_MLP::create

Constructs the MLP with the specified topology

```
void CvANN_MLP::create(
            const CvMat* _layer_sizes,
            int _activ_func=SIGMOID_SYM,
            double _f_param1=0,
            double _f_param2=0 );
```

_layer_sizes The integer vector specifies the number of neurons in each layer including the input and output layers.

_activ_func Specifies the activation function for each neuron; one of `CvANN_MLP::IDENTITY`, `CvANN_MLP::SIGMOID_SYM` and `CvANN_MLP::GAUSSIAN`.

_f_param1, _f_param2 Free parameters of the activation function, α and β, respectively. See the formulas in the introduction section.

The method creates a MLP network with the specified topology and assigns the same activation function to all the neurons.

CvANN_MLP::train

Trains/updates MLP.

```
int CvANN_MLP::train(
            const CvMat* _inputs,
            const CvMat* _outputs,
            const CvMat* _sample_weights,
            const CvMat* _sample_idx=0,
            CvANN_MLP_TrainParams _params = CvANN_MLP_TrainParams(),
            int flags=0 );
```

_inputs A floating-point matrix of input vectors, one vector per row.

_outputs A floating-point matrix of the corresponding output vectors, one vector per row.

_sample_weights (RPROP only) The optional floating-point vector of weights for each sample. Some samples may be more important than others for training, and the user may want to raise the weight of certain classes to find the right balance between hit-rate and false-alarm rate etc.

_sample_idx The optional integer vector indicating the samples (i.e. rows of `_inputs` and `_outputs`) that are taken into account.

_params The training params. See `CvANN_MLP_TrainParams` description.

_flags The various parameters to control the training algorithm. May be a combination of the following:

UPDATE_WEIGHTS = 1 algorithm updates the network weights, rather than computes them from scratch (in the latter case the weights are initialized using *Nguyen-Widrow* algorithm).

NO_INPUT_SCALE algorithm does not normalize the input vectors. If this flag is not set, the training algorithm normalizes each input feature independently, shifting its mean value to 0 and making the standard deviation =1. If the network is assumed to be updated frequently, the new training data could be much different from original one. In this case user should take care of proper normalization.

NO_OUTPUT_SCALE algorithm does not normalize the output vectors. If the flag is not set, the training algorithm normalizes each output features independently, by transforming it to the certain range depending on the activation function used.

This method applies the specified training algorithm to compute/adjust the network weights. It returns the number of done iterations.

Part III

Python API Reference

Chapter 12

cxcore. The Core Functionality

12.1 Basic Structures

CvPoint

2D point with integer coordinates (usually zero-based).

```
typedef struct CvPoint
{
    int x;
    int y;
}
CvPoint;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint cvPoint( int x, int y );

/* Conversion from CvPoint2D32f */
inline CvPoint cvPointFrom32f( CvPoint2D32f point );
```

CvPoint2D32f

2D point with floating-point coordinates

```
typedef struct CvPoint2D32f
{
    float x;
    float y;
}
CvPoint2D32f;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint2D32f cvPoint2D32f( double x, double y );

/* Conversion from CvPoint */
inline CvPoint2D32f cvPointTo32f( CvPoint point );
```

CvPoint3D32f

3D point with floating-point coordinates

```
typedef struct CvPoint3D32f
{
    float x;
    float y;
    float z;
}
CvPoint3D32f;
```

x x-coordinate

y y-coordinate

z z-coordinate

```
/* Constructor */
inline CvPoint3D32f cvPoint3D32f( double x, double y, double z );
```

CvPoint2D64f

2D point with double precision floating-point coordinates

```
typedef struct CvPoint2D64f
{
    double x;
    double y;
}
CvPoint2D64f;
```

x x-coordinate

y y-coordinate

```
/* Constructor */
inline CvPoint2D64f cvPoint2D64f( double x, double y );

/* Conversion from CvPoint */
inline CvPoint2D64f cvPointTo64f( CvPoint point );
```

CvPoint3D64f

3D point with double precision floating-point coordinates

```
typedef struct CvPoint3D64f
{
    double x;
    double y;
    double z;
}
CvPoint3D64f;
```

x x-coordinate

y y-coordinate

z z-coordinate

```
/* Constructor */
inline CvPoint3D64f cvPoint3D64f( double x, double y, double z );
```

CvSize

Pixel-accurate size of a rectangle.

Size of a rectangle, represented as a tuple `(width, height)`, where width and height are integers.

CvSize2D32f

Sub-pixel accurate size of a rectangle.

Size of a rectangle, represented as a tuple `(width, height)`, where width and height are floats.

CvRect

Offset (usually the top-left corner) and size of a rectangle.

```
typedef struct CvRect
{
    int x;
    int y;
    int width;
    int height;
}
CvRect;
```

x x-coordinate of the top-left corner

y y-coordinate of the top-left corner (bottom-left for Windows bitmaps)

width Width of the rectangle

height Height of the rectangle

```
/* Constructor */
inline CvRect cvRect( int x, int y, int width, int height );
```

CvScalar

A container for 1-,2-,3- or 4-tuples of doubles.

CvScalar is always represented as a 4-tuple.

```
>>> import cv
>>> cv.Scalar(1, 2, 3, 4)
(1.0, 2.0, 3.0, 4.0)
>>> cv.ScalarAll(7)
(7.0, 7.0, 7.0, 7.0)
>>> cv.RealScalar(7)
(7.0, 0.0, 0.0, 0.0)
>>> cv.RGB(0.1, 0.2, 0.3)
(0.29999999999999999, 0.20000000000000001, 0.10000000000000001, 0.0)
```

CvTermCriteria

Termination criteria for iterative algorithms.

Represented by a tuple `(type, max_iter, epsilon)`.

type `CV_TERMCRIT_ITER`, `CV_TERMCRIT_EPS` or `CV_TERMCRIT_ITER | CV_TERMCRIT_EPS`

max_iter Maximum number of iterations

epsilon Required accuracy

CvMat

A multi-channel matrix.

```
typedef struct CvMat
{
    int type;
    int step;

    int* refcount;

    union
    {
        uchar* ptr;
        short* s;
        int* i;
        float* fl;
        double* db;
    } data;

#ifdef __cplusplus
    union
```

```
    {
        int rows;
        int height;
    };

    union
    {
        int cols;
        int width;
    };
#else
    int rows;
    int cols;
#endif

} CvMat;
```

type A CvMat signature (CV_MAT_MAGIC_VAL) containing the type of elements and flags

step Full row length in bytes

refcount Underlying data reference counter

data Pointers to the actual matrix data

rows Number of rows

cols Number of columns

Matrices are stored row by row. All of the rows are aligned by 4 bytes.

CvMatND

Multi-dimensional dense multi-channel array.

```
typedef struct CvMatND
{
    int type;
    int dims;

    int* refcount;

    union
    {
        uchar* ptr;
```

```
        short* s;
        int* i;
        float* fl;
        double* db;
    } data;

    struct
    {
        int size;
        int step;
    }
    dim[CV_MAX_DIM];

} CvMatND;
```

type A CvMatND signature (CV_MATND_MAGIC_VAL), combining the type of elements and flags

dims The number of array dimensions

refcount Underlying data reference counter

data Pointers to the actual matrix data

dim For each dimension, the pair (number of elements, distance between elements in bytes)

CvSparseMat

Multi-dimensional sparse multi-channel array.

```
typedef struct CvSparseMat
{
    int type;
    int dims;
    int* refcount;
    struct CvSet* heap;
    void** hashtable;
    int hashsize;
    int total;
    int valoffset;
    int idxoffset;
    int size[CV_MAX_DIM];

} CvSparseMat;
```

type A CvSparseMat signature (CV_SPARSE_MAT_MAGIC_VAL), combining the type of elements and flags.

dims Number of dimensions

refcount Underlying reference counter. Not used.

heap A pool of hash table nodes

hashtable The hash table. Each entry is a list of nodes.

hashsize Size of the hash table

total Total number of sparse array nodes

valoffset The value offset of the array nodes, in bytes

idxoffset The index offset of the array nodes, in bytes

size Array of dimension sizes

IplImage

IPL image header

```
typedef struct _IplImage
{
    int  nSize;
    int  ID;
    int  nChannels;
    int  alphaChannel;
    int  depth;
    char colorModel[4];
    char channelSeq[4];
    int  dataOrder;
    int  origin;
    int  align;
    int  width;
    int  height;
    struct _IplROI *roi;
    struct _IplImage *maskROI;
    void  *imageId;
    struct _IplTileInfo *tileInfo;
    int  imageSize;
    char *imageData;
```

```
    int  widthStep;
    int  BorderMode[4];
    int  BorderConst[4];
    char *imageDataOrigin;
}
IplImage;
```

nSize sizeof(IplImage)

ID Version, always equals 0

nChannels Number of channels. Most OpenCV functions support 1-4 channels.

alphaChannel Ignored by OpenCV

depth Pixel depth in bits. The supported depths are:

- **IPL_DEPTH_8U** Unsigned 8-bit integer
- **IPL_DEPTH_8S** Signed 8-bit integer
- **IPL_DEPTH_16U** Unsigned 16-bit integer
- **IPL_DEPTH_16S** Signed 16-bit integer
- **IPL_DEPTH_32S** Signed 32-bit integer
- **IPL_DEPTH_32F** Single-precision floating point
- **IPL_DEPTH_64F** Double-precision floating point

colorModel Ignored by OpenCV. The OpenCV function CvtColor requires the source and destination color spaces as parameters.

channelSeq Ignored by OpenCV

dataOrder 0 = IPL_DATA_ORDER_PIXEL - interleaved color channels, 1 - separate color channels. CreateImage only creates images with interleaved channels. For example, the usual layout of a color image is: $b_{00}g_{00}r_{00}b_{10}g_{10}r_{10}...$

origin 0 - top-left origin, 1 - bottom-left origin (Windows bitmap style)

align Alignment of image rows (4 or 8). OpenCV ignores this and uses widthStep instead.

width Image width in pixels

height Image height in pixels

roi Region Of Interest (ROI). If not NULL, only this image region will be processed.

maskROI Must be NULL in OpenCV

imageId Must be NULL in OpenCV

tileInfo Must be NULL in OpenCV

imageSize Image data size in bytes. For interleaved data, this equals `image->height·image->widthStep`

imageData A pointer to the aligned image data

widthStep The size of an aligned image row, in bytes

BorderMode Border completion mode, ignored by OpenCV

BorderConst Border completion mode, ignored by OpenCV

imageDataOrigin A pointer to the origin of the image data (not necessarily aligned). This is used for image deallocation.

The IplImage structure was inherited from the Intel Image Processing Library, in which the format is native. OpenCV only supports a subset of possible IplImage formats, as outlined in the parameter list above.

In addition to the above restrictions, OpenCV handles ROIs differently. OpenCV functions require that the image size or ROI size of all source and destination images match exactly. On the other hand, the Intel Image Processing Library processes the area of intersection between the source and destination images (or ROIs), allowing them to vary independently.

CvArr

Arbitrary array

```
typedef void CvArr;
```

The metatype `CvArr` is used *only* as a function parameter to specify that the function accepts arrays of multiple types, such as IplImage*, CvMat* or even CvSeq* sometimes. The particular array type is determined at runtime by analyzing the first 4 bytes of the header.

12.2 Operations on Arrays

---

cv.AbsDiff

Calculates absolute difference between two arrays.

```
AbsDiff(src1,src2,dst)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates absolute difference between two arrays.

$$\texttt{dst}(i)_c = |\texttt{src1}(I)_c - \texttt{src2}(I)_c|$$

All the arrays must have the same data type and the same size (or ROI size).

---

cv.AbsDiffS

Calculates absolute difference between an array and a scalar.

```
AbsDiffS(src,value,dst)-> None
```

src The source array

dst The destination array

value The scalar

The function calculates absolute difference between an array and a scalar.

$$\texttt{dst}(i)_c = |\texttt{src}(I)_c - \texttt{value}_c|$$

All the arrays must have the same data type and the same size (or ROI size).

---

cv.Add

Computes the per-element sum of two arrays.

```
Add(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds one array to another:

```
dst(I)=src1(I)+src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cv.AddS

Computes the sum of an array and a scalar.

```
AddS(src,value,dst,mask=NULL)-> None
```

src The source array

value Added scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function adds a scalar `value` to every element in the source array `src1` and stores the result in `dst`. For types that have limited range this operation is saturating.

```
dst(I)=src(I)+value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size).

cv.AddWeighted

Computes the weighted sum of two arrays.

```
AddWeighted(src1,alpha,src2,beta,gamma,dst)-> None
```

src1 The first source array

alpha Weight for the first array elements

src2 The second source array

beta Weight for the second array elements

dst The destination array

gamma Scalar, added to each sum

The function calculates the weighted sum of two arrays as follows:

```
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
```

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

cv.And

Calculates per-element bit-wise conjunction of two arrays.

```
And(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)&src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.AndS

Calculates per-element bit-wise conjunction of an array and a scalar.

```
AndS(src,value,dst,mask=NULL)-> None
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)&value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

The following sample demonstrates how to calculate the absolute value of floating-point array elements by clearing the most-significant bit:

```
float a[] = { -1, 2, -3, 4, -5, 6, -7, 8, -9 };
CvMat A = cvMat(3, 3, CV\_32F, &a);
int i, absMask = 0x7fffffff;
cvAndS(&A, cvRealScalar(*(float*)&absMask), &A, 0);
for(i = 0; i < 9; i++ )
    printf("%.1f ", a[i]);
```

The code should print:

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

cv.Avg

Calculates average (mean) of array elements.

```
Avg(arr,mask=NULL)-> CvScalar
```

arr The array

mask The optional operation mask

The function calculates the average value `M` of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$M_c = \frac{\sum_{I,\texttt{mask}(I)\neq 0} \texttt{arr}(I)_c}{N}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average to the first scalar component S_0 .

cv.AvgSdv

Calculates average (mean) of array elements.

```
AvgSdv(arr,mask=NULL)-> (mean, stdDev)
```

arr The array

mask The optional operation mask

mean Mean value, a CvScalar

stdDev Standard deviation, a CvScalar

The function calculates the average value and standard deviation of array elements, independently for each channel:

$$N = \sum_I (\texttt{mask}(I) \neq 0)$$
$$mean_c = \frac{1}{N} \sum_{I,\texttt{mask}(I)\neq 0} \texttt{arr}(I)_c$$
$$stdDev_c = \sqrt{\frac{1}{N} \sum_{I,\texttt{mask}(I)\neq 0} (\texttt{arr}(I)_c - mean_c)^2}$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the average and standard deviation to the first components of the output scalars ($mean_0$ and $stdDev_0$).

---

cv.CalcCovarMatrix

Calculates covariance matrix of a set of vectors.

```
CalcCovarMatrix(vects,covMat,avg,flags)-> None
```

vects The input vectors, all of which must have the same type and the same size. The vectors do not have to be 1D, they can be 2D (e.g., images) and so forth

covMat The output covariance matrix that should be floating-point and square

avg The input or output (depending on the flags) array - the mean (average) vector of the input vectors

flags The operation flags, a combination of the following values

CV_COVAR_SCRAMBLED The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]$$

, that is, the covariance matrix is `count` $\times$ `count`. Such an unusual covariance matrix is used for fast PCA of a set of very large vectors (see, for example, the EigenFaces technique for face recognition). Eigenvalues of this "scrambled" matrix will match the eigenvalues of the true covariance matrix and the "true" eigenvectors can be easily calculated from the eigenvectors of the "scrambled" covariance matrix.

CV_COVAR_NORMAL The output covariance matrix is calculated as:

$$\texttt{scale} * [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...] \cdot [\texttt{vects}[0] - \texttt{avg}, \texttt{vects}[1] - \texttt{avg}, ...]^T$$

, that is, `covMat` will be a covariance matrix with the same linear size as the total number of elements in each input vector. One and only one of `CV_COVAR_SCRAMBLED` and `CV_COVAR_NORMAL` must be specified

CV_COVAR_USE_AVG If the flag is specified, the function does not calculate `avg` from the input vectors, but, instead, uses the passed `avg` vector. This is useful if `avg` has been already calculated somehow, or if the covariance matrix is calculated by parts - in this case, `avg` is not a mean vector of the input sub-set of vectors, but rather the mean vector of the whole set.

CV_COVAR_SCALE If the flag is specified, the covariance matrix is scaled. In the "normal" mode `scale` is '1./count'; in the "scrambled" mode `scale` is the reciprocal of the total number of elements in each input vector. By default (if the flag is not specified) the covariance matrix is not scaled ('scale=1').

CV_COVAR_ROWS Means that all the input vectors are stored as rows of a single matrix, `vects[0]. count` is ignored in this case, and `avg` should be a single-row vector of an appropriate size.

CV_COVAR_COLS Means that all the input vectors are stored as columns of a single matrix, `vects[0]. count` is ignored in this case, and `avg` should be a single-column vector of an appropriate size.

The function calculates the covariance matrix and, optionally, the mean vector of the set of input vectors. The function can be used for PCA, for comparing vectors using Mahalanobis distance and so forth.

---

cv.CartToPolar

Calculates the magnitude and/or angle of 2d vectors.

```
CartToPolar(x,y,magnitude,angle=NULL,angleInDegrees=0)-> None
```

x The array of x-coordinates

y The array of y-coordinates

magnitude The destination array of magnitudes, may be set to NULL if it is not needed

angle The destination array of angles, may be set to NULL if it is not needed. The angles are measured in radians $(0$ to $2\pi)$ or in degrees (0 to 360 degrees).

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

```
magnitude(I)=sqrt(x(I)^2^+y(I)^2^ ),
angle(I)=atan(y(I)/x(I) )
```

The angles are calculated with 0.1 degree accuracy. For the (0,0) point, the angle is set to 0.

cv.Cbrt

Calculates the cubic root

```
Cbrt(value)-> float
```

value The input floating-point value

The function calculates the cubic root of the argument, and normally it is faster than `pow(value,1./3)`. In addition, negative arguments are handled properly. Special values ($\pm\infty$, NaN) are not handled.

cv.ClearND

Clears a specific array element.

```
ClearND(arr,idx)-> None
```

arr Input array

idx Array of the element indices

The function cv.ClearND clears (sets to zero) a specific element of a dense array or deletes the element of a sparse array. If the sparse array element does not exists, the function does nothing.

cv.CloneImage

Makes a full copy of an image, including the header, data, and ROI.

```
CloneImage(image)-> copy
```

image The original image

The returned `IplImage*` points to the image copy.

cv.CloneMat

Creates a full matrix copy.

```
CloneMat(mat)-> copy
```

mat Matrix to be copied

Creates a full copy of a matrix and returns a pointer to the copy.

cv.CloneMatND

Creates full copy of a multi-dimensional array and returns a pointer to the copy.

```
CloneMatND(mat)-> copy
```

mat Input array

cv.Cmp

Performs per-element comparison of two arrays.

```
Cmp(src1,src2,dst,cmpOp)-> None
```

src1 The first source array

src2 The second source array. Both source arrays must have a single channel.

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

CV_CMP_EQ src1(I) ”equal to” value

CV_CMP_GT src1(I) ”greater than” value

CV_CMP_GE src1(I) ”greater or equal” value

CV_CMP_LT src1(I) "less than" value

CV_CMP_LE src1(I) "less or equal" value

CV_CMP_NE src1(I) "not equal" value

The function compares the corresponding elements of two arrays and fills the destination mask array:

```
dst(I)=src1(I) op src2(I),
```

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size)

---

cv.CmpS

Performs per-element comparison of an array and a scalar.

```
CmpS(src,value,dst,cmpOp)-> None
```

src The source array, must have a single channel

value The scalar value to compare each array element with

dst The destination array, must have 8u or 8s type

cmpOp The flag specifying the relation between the elements to be checked

CV_CMP_EQ src1(I) "equal to" value

CV_CMP_GT src1(I) "greater than" value

CV_CMP_GE src1(I) "greater or equal" value

CV_CMP_LT src1(I) "less than" value

CV_CMP_LE src1(I) "less or equal" value

CV_CMP_NE src1(I) "not equal" value

The function compares the corresponding elements of an array and a scalar and fills the destination mask array:

```
dst(I)=src(I) op scalar
```

where `op` is $=$, $>$, $\geq$, $<$, $\leq$ or $\neq$.

`dst(I)` is set to 0xff (all `1`-bits) if the specific relation between the elements is true and 0 otherwise. All the arrays must have the same size (or ROI size).

cv.Convert

Converts one array to another.

```
Convert(src,dst)-> None
```

src Source array

dst Destination array

The type of conversion is done with rounding and saturation, that is if the result of scaling + conversion can not be represented exactly by a value of the destination array element type, it is set to the nearest representable value on the real axis.

All the channels of multi-channel arrays are processed independently.

cv.ConvertScale

Converts one array to another with optional linear transformation.

```
ConvertScale(src,dst,scale=1.0,shift=0.0)-> None
```

src Source array

dst Destination array

scale Scale factor

shift Value added to the scaled source array elements

The function has several different purposes, and thus has several different names. It copies one array to another with optional scaling, which is performed first, and/or optional type conversion, performed after:

$$\texttt{dst}(I) = \texttt{scale}\texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)$$

All the channels of multi-channel arrays are processed independently.

The type of conversion is done with rounding and saturation, that is if the result of scaling + conversion can not be represented exactly by a value of the destination array element type, it is set to the nearest representable value on the real axis.

In the case of `scale=1, shift=0` no prescaling is done. This is a specially optimized case and it has the appropriate cv.Convert name. If source and destination array types have equal types, this is also a special case that can be used to scale and shift a matrix or an image and that is caled cv.Scale.

---

cv.ConvertScaleAbs

Converts input array elements to another 8-bit unsigned integer with optional linear transformation.

```
ConvertScaleAbs(src,dst,scale=1.0,shift=0.0)-> None
```

```
#define cvCvtScaleAbs cvConvertScaleAbs
```

src Source array

dst Destination array (should have 8u depth)

scale ScaleAbs factor

shift Value added to the scaled source array elements

The function is similar to cv.ConvertScale, but it stores absolute values of the conversion results:

$$\texttt{dst}(I) = |\texttt{scale}\texttt{src}(I) + (\texttt{shift}_0, \texttt{shift}_1, ...)|$$

The function supports only destination arrays of 8u (8-bit unsigned integers) type; for other types the function can be emulated by a combination of cv.ConvertScale and cv.Abs functions.

---

cv.Copy

Copies one array to another.

```
Copy(src,dst,mask=NULL)-> None
```

src The source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies selected elements from an input array to an output array:

$$\texttt{dst}(I) = \texttt{src}(I) \quad \text{if} \quad \texttt{mask}(I) \neq 0.$$

If any of the passed arrays is of `IplImage` type, then its ROI and COI fields are used. Both arrays must have the same type, the same number of dimensions, and the same size. The function can also copy sparse arrays (mask is not supported in this case).

---

cv.CountNonZero

Counts non-zero array elements.

```
CountNonZero(arr)-> int
```

arr The array must be a single-channel array or a multi-channel image with COI set

The function returns the number of non-zero elements in arr:

$$\sum_I (\texttt{arr}(I) \neq 0)$$

In the case of `IplImage` both ROI and COI are supported.

---

cv.CreateData

Allocates array data

```
CreateData(arr) -> None
```

arr Array header

The function allocates image, matrix or multi-dimensional array data. Note that in the case of matrix types OpenCV allocation functions are used and in the case of IplImage they are used unless `CV_TURN_ON_IPL_COMPATIBILITY` was called. In the latter case IPL functions are used to allocate the data.

cv.CreateImage

Creates an image header and allocates the image data.

```
CreateImage(size, depth, channels)->image
```

size Image width and height

depth Bit depth of image elements. See IplImage for valid depths.

channels Number of channels per pixel. See IplImage for details. This function only creates images with interleaved channels.

This call is a shortened form of

```
header = cvCreateImageHeader(size, depth, channels);
cvCreateData(header);
```

cv.CreateImageHeader

Creates an image header but does not allocate the image data.

```
CreateImageHeader(size, depth, channels) -> image
```

size Image width and height

depth Image depth (see cv.CreateImage)

channels Number of channels (see cv.CreateImage)

This call is an analogue of

```
hdr=iplCreateImageHeader(channels, 0, depth,
                         channels == 1 ? "GRAY" : "RGB",
                         channels == 1 ? "GRAY" : channels == 3 ? "BGR" :
                         channels == 4 ? "BGRA" : "",
                         IPL_DATA_ORDER_PIXEL, IPL_ORIGIN_TL, 4,
                         size.width, size.height,
                         0,0,0,0);
```

but it does not use IPL functions by default (see the `CV_TURN_ON_IPL_COMPATIBILITY` macro).

cv.CreateMat

Creates a matrix header and allocates the matrix data.

```
CreateMat(rows, cols, type) -> mat
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type The type of the matrix elements in the form `CV_<bit depth><S|U|F>C<number of channels>`, where S=signed, U=unsigned, F=float. For example, CV_8UC1 means the elements are 8-bit unsigned and the there is 1 channel, and CV_32SC2 means the elements are 32-bit signed and there are 2 channels.

This is the concise form for:

```
CvMat* mat = cvCreateMatHeader(rows, cols, type);
cvCreateData(mat);
```

cv.CreateMatHeader

Creates a matrix header but does not allocate the matrix data.

```
CreateMatHeader(rows, cols, type) -> mat
```

rows Number of rows in the matrix

cols Number of columns in the matrix

type Type of the matrix elements, see cv.CreateMat

The function allocates a new matrix header and returns a pointer to it. The matrix data can then be allocated using cv.CreateData or set explicitly to user-allocated data via cv.SetData.

cv.CreateMatND

Creates the header and allocates the data for a multi-dimensional dense array.

```
CreateMatND(dims, type) -> None
```

dims List or tuple of array dimensions, up to 32 in length.

type Type of array elements, see cv.CreateMat.

This is a short form for:

```
CvMatND* mat = cvCreateMatNDHeader(dims, sizes, type);
cvCreateData(mat);
```

---

cv.CreateMatNDHeader

Creates a new matrix header but does not allocate the matrix data.

```
CreateMatNDHeader(dims, type) -> None
```

dims List or tuple of array dimensions, up to 32 in length.

type Type of array elements, see cv.CreateMat

The function allocates a header for a multi-dimensional dense array. The array data can further be allocated using cv.CreateData or set explicitly to user-allocated data via cv.SetData.

---

cv.CrossProduct

Calculates the cross product of two 3D vectors.

```
CrossProduct(src1,src2,dst)-> None
```

src1 The first source vector

src2 The second source vector

dst The destination vector

The function calculates the cross product of two 3D vectors:

$$\texttt{dst} = \texttt{src1} \times \texttt{src2}$$

or:

$$\begin{aligned}\texttt{dst}_1 &= \texttt{src1}_2\texttt{src2}_3 - \texttt{src1}_3\texttt{src2}_2 \\ \texttt{dst}_2 &= \texttt{src1}_3\texttt{src2}_1 - \texttt{src1}_1\texttt{src2}_3 \\ \texttt{dst}_3 &= \texttt{src1}_1\texttt{src2}_2 - \texttt{src1}_2\texttt{src2}_1\end{aligned}$$

---

cv.DCT

Performs a forward or inverse Discrete Cosine transform of a 1D or 2D floating-point array.

```
DCT(src,dst,flags)-> None
```

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_ROWS     4
```

src Source array, real 1D or 2D array

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

CV_DXT_FORWARD do a forward 1D or 2D transform.

CV_DXT_INVERSE do an inverse 1D or 2D transform.

CV_DXT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

The function performs a forward or inverse transform of a 1D or 2D floating-point array: Forward Cosine transform of 1D vector of N elements:

$$Y = C^{(N)} \cdot X$$

where

$$C_{jk}^{(N)} = \sqrt{\alpha_j/N} \cos\left(\frac{\pi(2k+1)j}{2N}\right)$$

and $\alpha_0 = 1$, $\alpha_j = 2$ for $j > 0$.

Inverse Cosine transform of 1D vector of N elements:

$$X = \left(C^{(N)}\right)^{-1} \cdot Y = \left(C^{(N)}\right)^{T} \cdot Y$$

(since $C^{(N)}$ is orthogonal matrix, $C^{(N)} \cdot \left(C^{(N)}\right)^{T} = I$)

Forward Cosine transform of 2D $M \times N$ matrix:

$$Y = C^{(N)} \cdot X \cdot \left(C^{(N)}\right)^{T}$$

Inverse Cosine transform of 2D vector of $M \times N$ elements:

$$X = \left(C^{(N)}\right)^{T} \cdot X \cdot C^{(N)}$$

---

cv.DFT

Performs a forward or inverse Discrete Fourier transform of a 1D or 2D floating-point array.

```
DFT(src,dst,flags,nonzeroRows=0)-> None
```

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_SCALE    2
#define CV_DXT_ROWS     4
#define CV_DXT_INV_SCALE (CV_DXT_SCALE|CV_DXT_INVERSE)
#define CV_DXT_INVERSE_SCALE CV_DXT_INV_SCALE
```

src Source array, real or complex

dst Destination array of the same size and same type as the source

flags Transformation flags, a combination of the following values

CV_DXT_FORWARD do a forward 1D or 2D transform. The result is not scaled.

CV_DXT_INVERSE do an inverse 1D or 2D transform. The result is not scaled. CV_DXT_FORWARD and CV_DXT_INVERSE are mutually exclusive, of course.

CV_DXT_SCALE scale the result: divide it by the number of array elements. Usually, it is combined with CV_DXT_INVERSE, and one may use a shortcut CV_DXT_INV_SCALE.

CV_DXT_ROWS do a forward or inverse transform of every individual row of the input matrix. This flag allows the user to transform multiple vectors simultaneously and can be used to decrease the overhead (which is sometimes several times larger than the processing itself), to do 3D and higher-dimensional transforms and so forth.

nonzeroRows Number of nonzero rows in the source array (in the case of a forward 2d transform), or a number of rows of interest in the destination array (in the case of an inverse 2d transform). If the value is negative, zero, or greater than the total number of rows, it is ignored. The parameter can be used to speed up 2d convolution/correlation when computing via DFT. See the example below.

The function performs a forward or inverse transform of a 1D or 2D floating-point array:
Forward Fourier transform of 1D vector of N elements:

$$y = F^{(N)} \cdot x, where F^{(N)}_{jk} = exp(-i \cdot 2\pi \cdot j \cdot k/N)$$

,

$$i = sqrt(-1)$$

Inverse Fourier transform of 1D vector of N elements:

$$x' = (F^{(N)})^{-1} \cdot y = conj(F^{(}N)) \cdot yx = (1/N) \cdot x$$

Forward Fourier transform of 2D vector of M $\times$ N elements:

$$Y = F^{(M)} \cdot X \cdot F^{(N)}$$

Inverse Fourier transform of 2D vector of M $\times$ N elements:

$$X' = conj(F^{(M)}) \cdot Y \cdot conj(F^{(N)})X = (1/(M \cdot N)) \cdot X'$$

In the case of real (single-channel) data, the packed format, borrowed from IPL, is used to represent the result of a forward Fourier transform or input for an inverse Fourier transform:

$$\begin{bmatrix} ReY_{0,0} & ReY_{0,1} & ImY_{0,1} & ReY_{0,2} & ImY_{0,2} & \cdots & ReY_{0,N/2-1} & ImY_{0,N/2-1} & ReY_{0,N/2} \\ ReY_{1,0} & ReY_{1,1} & ImY_{1,1} & ReY_{1,2} & ImY_{1,2} & \cdots & ReY_{1,N/2-1} & ImY_{1,N/2-1} & ReY_{1,N/2} \\ ImY_{1,0} & ReY_{2,1} & ImY_{2,1} & ReY_{2,2} & ImY_{2,2} & \cdots & ReY_{2,N/2-1} & ImY_{2,N/2-1} & ImY_{1,N/2} \\ \hdots & & & & & & & & \\ ReY_{M/2-1,0} & ReY_{M-3,1} & ImY_{M-3,1} & \hdots & & & ReY_{M-3,N/2-1} & ImY_{M-3,N/2-1} & ReY_{M/2-1,N/2} \\ ImY_{M/2-1,0} & ReY_{M-2,1} & ImY_{M-2,1} & \hdots & & & ReY_{M-2,N/2-1} & ImY_{M-2,N/2-1} & ImY_{M/2-1,N/2} \\ ReY_{M/2,0} & ReY_{M-1,1} & ImY_{M-1,1} & \hdots & & & ReY_{M-1,N/2-1} & ImY_{M-1,N/2-1} & ReY_{M/2,N/2} \end{bmatrix}$$

Note: the last column is present if `N` is even, the last row is present if `M` is even. In the case of 1D real transform the result looks like the first row of the above matrix.

Here is the example of how to compute 2D convolution using DFT.

```
CvMat* A = cvCreateMat(M1, N1, CVg32F);
CvMat* B = cvCreateMat(M2, N2, A->type);

// it is also possible to have only abs(M2-M1)+1 times abs(N2-N1)+1
// part of the full convolution result
CvMat* conv = cvCreateMat(A->rows + B->rows - 1, A->cols + B->cols - 1,
                          A->type);

// initialize A and B
...

int dftgM = cvGetOptimalDFTSize(A->rows + B->rows - 1);
int dftgN = cvGetOptimalDFTSize(A->cols + B->cols - 1);

CvMat* dftgA = cvCreateMat(dft\_M, dft\_N, A->type);
CvMat* dftgB = cvCreateMat(dft\_M, dft\_N, B->type);
CvMat tmp;

// copy A to dftgA and pad dft\_A with zeros
cvGetSubRect(dftgA, &tmp, cvRect(0,0,A->cols,A->rows));
cvCopy(A, &tmp);
cvGetSubRect(dftgA, &tmp, cvRect(A->cols,0,dft\_A->cols - A->cols,A->rows));
cvZero(&tmp);
// no need to pad bottom part of dftgA with zeros because of
// use nonzerogrows parameter in cvDFT() call below

cvDFT(dftgA, dft\_A, CV\_DXT\_FORWARD, A->rows);

// repeat the same with the second array
cvGetSubRect(dftgB, &tmp, cvRect(0,0,B->cols,B->rows));
cvCopy(B, &tmp);
cvGetSubRect(dftgB, &tmp, cvRect(B->cols,0,dft\_B->cols - B->cols,B->rows));
cvZero(&tmp);
// no need to pad bottom part of dftgB with zeros because of
// use nonzerogrows parameter in cvDFT() call below

cvDFT(dftgB, dft\_B, CV\_DXT\_FORWARD, B->rows);

cvMulSpectrums(dftgA, dft\_B, dft\_A, 0 /* or CV\_DXT\_MUL\_CONJ to get
               correlation rather than convolution */);

cvDFT(dftgA, dft\_A, CV\_DXT\_INV\_SCALE, conv->rows); // calculate only
                                                      // the top part
cvGetSubRect(dftgA, &tmp, cvRect(0,0,conv->cols,conv->rows));
```

```
cvCopy(&tmp, conv);
```

---

cv.Det

Returns the determinant of a matrix.

```
Det(mat)-> double
```

mat The source matrix

The function returns the determinant of the square matrix `mat`. The direct method is used for small matrices and Gaussian elimination is used for larger matrices. For symmetric positive-determined matrices, it is also possible to run cv.SVD with $U = V = 0$ and then calculate the determinant as a product of the diagonal elements of W.

---

cv.Div

Performs per-element division of two arrays.

```
Div(src1,src2,dst,scale)-> None
```

src1 The first source array. If the pointer is NULL, the array is assumed to be all 1's.

src2 The second source array

dst The destination array

scale Optional scale factor

The function divides one array by another:

$$\texttt{dst}(I) = \begin{cases} \texttt{scale} \cdot \texttt{src1}(I)/\texttt{src2}(I) & \text{if } \texttt{src1} \text{ is not } \texttt{NULL} \\ \texttt{scale}/\texttt{src2}(I) & \text{otherwise} \end{cases}$$

All the arrays must have the same type and the same size (or ROI size).

cv.DotProduct

Calculates the dot product of two arrays in Euclidian metrics.

```
DotProduct(src1,src2)-> double
```

src1 The first source array

src2 The second source array

The function calculates and returns the Euclidean dot product of two arrays.

$$src1 \bullet src2 = \sum_I (\texttt{src1}(I)\texttt{src2}(I))$$

In the case of multiple channel arrays, the results for all channels are accumulated. In particular, `cvDotProduct(a,a)` where `a` is a complex vector, will return $||\texttt{a}||^2$. The function can process multi-dimensional arrays, row by row, layer by layer, and so on.

cv.EigenVV

Computes eigenvalues and eigenvectors of a symmetric matrix.

```
EigenVV(mat,evects,evals,eps,lowindex,highindex)-> None
```

mat The input symmetric square matrix, modified during the processing

evects The output matrix of eigenvectors, stored as subsequent rows

evals The output vector of eigenvalues, stored in the descending order (order of eigenvalues and eigenvectors is syncronized, of course)

eps Accuracy of diagonalization. Typically, `DBL_EPSILON` (about 10^{-15}) works well. THIS PARAMETER IS CURRENTLY IGNORED.

lowindex Optional index of largest eigenvalue/-vector to calculate. (See below.)

highindex Optional index of smallest eigenvalue/-vector to calculate. (See below.)

The function computes the eigenvalues and eigenvectors of matrix A:

```
mat*evects(i,:)' = evals(i)*evects(i,:)' (in MATLAB notation)
```

If either low- or highindex is supplied the other is required, too. Indexing is 1-based. Example: To calculate the largest eigenvector/-value set lowindex = highindex = 1. For legacy reasons this function always returns a square matrix the same size as the source matrix with eigenvectors and a vector the length of the source matrix with eigenvalues. The selected eigenvectors/-values are always in the first highindex - lowindex + 1 rows.

The contents of matrix A is destroyed by the function.

Currently the function is slower than cv.SVD yet less accurate, so if A is known to be positively-defined (for example, it is a covariance matrix)it is recommended to use cv.SVD to find eigenvalues and eigenvectors of A, especially if eigenvectors are not required.

---

cv.Exp

Calculates the exponent of every array element.

```
Exp(src,dst)-> None
```

src The source array

dst The destination array, it should have double type or the same type as the source

The function calculates the exponent of every element of the input array:

$$\texttt{dst}[I] = e^{\texttt{src}(I)}$$

The maximum relative error is about 7×10^{-6}. Currently, the function converts denormalized values to zeros on output.

---

cv.FastArctan

Calculates the angle of a 2D vector.

```
FastArctan(y,x)-> float
```

x x-coordinate of 2D vector

y y-coordinate of 2D vector

The function calculates the full-range angle of an input 2D vector. The angle is measured in degrees and varies from 0 degrees to 360 degrees. The accuracy is about 0.1 degrees.

---

cv.Flip

Flip a 2D array around vertical, horizontal or both axes.

```
Flip(src,dst=NULL,flipMode=0)-> None
```

```
#define cvMirror cvFlip
```

src Source array

dst Destination array. If $\texttt{dst} = \texttt{NULL}$ the flipping is done in place.

flipMode Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas:

The function flips the array in one of three different ways (row and column indices are 0-based):

$$dst(i,j) = \begin{cases} \texttt{src}(rows(\texttt{src}) - i - 1, j) & \text{if } \texttt{flipMode} = 0 \\ \texttt{src}(i, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} > 0 \\ \texttt{src}(rows(\texttt{src}) - i - 1, cols(\texttt{src}) - j - 1) & \text{if } \texttt{flipMode} < 0 \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (flipMode = 0) to switch between top-left and bottom-left image origin, which is a typical operation in video processing under Win32 systems.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (flipMode $>$ 0)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flipMode $<$ 0)
- reversing the order of 1d point arrays (flipMode ¿ 0)

cv.GEMM

Performs generalized matrix multiplication.

```
GEMM(src1,src2,alphs,src3,beta,dst,tABC=0)-> None
```

src1 The first source array

src2 The second source array

src3 The third source array (shift). Can be NULL, if there is no shift.

dst The destination array

tABC The operation flags that can be 0 or a combination of the following values

CV_GEMM_A_T transpose src1
CV_GEMM_B_T transpose src2
CV_GEMM_C_T transpose src3

For example, CV_GEMM_A_T+CV_GEMM_C_T corresponds to

$$\texttt{alpha}\,\texttt{src1}^T\,\texttt{src2} + \texttt{beta}\,\texttt{src3}^T$$

The function performs generalized matrix multiplication:

$$\texttt{dst} = \texttt{alpha}\,op(\texttt{src1})\,op(\texttt{src2}) + \texttt{beta}\,op(\texttt{src3}) \quad \text{where } op(X) \text{ is } X \text{ or } X^T$$

All the matrices should have the same data type and coordinated sizes. Real or complex floating-point matrices are supported.

cv.Get1D

Return a specific array element.

```
Get1D(arr, idx) -> scalar
```

arr Input array

idx Zero-based element index

Return a specific array element. Array must have dimension 3.

cv.Get2D

Return a specific array element.

```
Get2D(arr, idx0, idx1) -> scalar
```

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

Return a specific array element. Array must have dimension 2.

cv.Get3D

Return a specific array element.

```
Get3D(arr, idx0, idx1, idx2) -> scalar
```

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

Return a specific array element. Array must have dimension 3.

cv.GetND

Return a specific array element.

```
GetND(arr, indices) -> scalar
```

arr Input array

indices List of zero-based element indices

Return a specific array element. The length of array indices must be the same as the dimension of the array.

---

cv.GetCol

Returns array column.

```
GetCol(arr,col)-> submat
```

arr Input array

col Zero-based index of the selected column

submat resulting single-column array

The function `GetCol` returns a single column from the input array.

---

cv.GetCols

Returns array column span.

```
GetCols(arr,startCol,endCol)-> submat
```

arr Input array

startCol Zero-based index of the starting column (inclusive) of the span

endCol Zero-based index of the ending column (exclusive) of the span

submat resulting multi-column array

The function `GetCols` returns a column span from the input array.

cv.GetDiag

Returns one of array diagonals.

```
GetDiag(arr,diag=0)-> submat
```

arr Input array

submat Pointer to the resulting sub-array header

diag Array diagonal. Zero corresponds to the main diagonal, -1 corresponds to the diagonal above the main , 1 corresponds to the diagonal below the main, and so forth.

The function returns the header, corresponding to a specified diagonal of the input array.

cv.GetDims

Returns list of array dimensions

```
GetDims(arr)-> list
```

arr Input array

The function returns a list of array dimensions. In the case of `IplImage` or CvMat it always returns a list of length 2.

cv.GetElemType

Returns type of array elements.

```
GetElemType(arr)-> int
```

arr Input array

The function returns type of the array elements as described in cv.CreateMat discussion: `CV_8UC1` ... `CV_64FC4`.

cv.GetImage

Returns image header for arbitrary array.

```
GetImage(arr) -> iplimage
```

arr Input array

The function returns the image header for the input array that can be a matrix - CvMat , or an image - `IplImage*`. In the case of an image the function simply returns the input pointer. In the case of CvMat it initializes an `imageHeader` structure with the parameters of the input matrix. Note that if we transform `IplImage` to CvMat and then transform CvMat back to IplImage, we can get different headers if the ROI is set, and thus some IPL functions that calculate image stride from its width and align may fail on the resultant image.

cv.GetImageCOI

Returns the index of the channel of interest.

```
GetImageCOI(image)-> channel
```

image A pointer to the image header

Returns the channel of interest of in an IplImage. Returned values correspond to the `coi` in cv.SetImageCOI.

cv.GetImageROI

Returns the image ROI.

```
GetImageROI(image)-> CvRect
```

image A pointer to the image header

If there is no ROI set, `cvRect(0,0,image->width,image->height)` is returned.

cv.GetMat

Returns matrix header for arbitrary array.

```
GetMat(arr) -> cvmat
```

arr Input array

The function returns a matrix header for the input array that can be a matrix - CvMat , an image - IplImage or a multi-dimensional dense array - CvMatND (latter case is allowed only if allowND != 0) . In the case of matrix the function simply returns the input pointer. In the case of IplImage* or CvMatND it initializes the header structure with parameters of the current image ROI and returns the pointer to this temporary structure. Because COI is not supported by CvMat , it is returned separately.

The function provides an easy way to handle both types of arrays - IplImage and CvMat - using the same code. Reverse transform from CvMat to IplImage can be done using the cv.GetImage function.

Input array must have underlying data allocated or attached, otherwise the function fails.

If the input array is IplImage with planar data layout and COI set, the function returns the pointer to the selected plane and COI = 0. It enables per-plane processing of multi-channel images with planar data layout using OpenCV functions.

cv.GetOptimalDFTSize

Returns optimal DFT size for a given vector size.

```
GetOptimalDFTSize(size0)-> int
```

size0 Vector size

The function returns the minimum number N that is greater than or equal to size0, such that the DFT of a vector of size N can be computed fast. In the current implementation $N = 2^p \times 3^q \times 5^r$, for some p, q, r.

The function returns a negative number if size0 is too large (very close to INT_MAX)

cv.GetRow

Returns array row.

```
GetRow(arr,row)-> submat
```

arr Input array

row Zero-based index of the selected row

submat resulting single-row array

The function `GetRow` returns a single row from the input array.

cv.GetRows

Returns array row span.

```
GetRows(arr,startRow,endRow,deltaRow=1)-> submat
```

arr Input array

startRow Zero-based index of the starting row (inclusive) of the span

endRow Zero-based index of the ending row (exclusive) of the span

deltaRow Index step in the row span.

submat resulting multi-row array

The function `GetRows` returns a row span from the input array.

cv.GetSize

Returns size of matrix or image ROI.

```
GetSize(arr)-> CvSize
```

arr array header

The function returns number of rows (CvSize::height) and number of columns (CvSize::width) of the input matrix or image. In the case of image the size of ROI is returned.

---

cv.GetSubRect

Returns matrix header corresponding to the rectangular sub-array of input image or matrix.

```
GetSubRect(arr, rect) -> cvmat
```

arr Input array

rect Zero-based coordinates of the rectangle of interest

The function returns header, corresponding to a specified rectangle of the input array. In other words, it allows the user to treat a rectangular part of input array as a stand-alone array. ROI is taken into account by the function so the sub-array of ROI is actually extracted.

---

cv.InRange

Checks that array elements lie between the elements of two other arrays.

```
InRange(src,lower,upper,dst)-> None
```

src The first source array

lower The inclusive lower boundary array

upper The exclusive upper boundary array

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}(I)_0 <= \texttt{src}(I)_0 < \texttt{upper}(I)_0 \wedge \texttt{lower}(I)_1 <= \texttt{src}(I)_1 < \texttt{upper}(I)_1$$

For two-channel arrays and so forth,
dst(I) is set to 0xff (all 1-bits) if src(I) is within the range and 0 otherwise. All the arrays must have the same type, except the destination, and the same size (or ROI size).

---

cv.InRangeS

Checks that array elements lie between two scalars.

```
InRangeS(src,lower,upper,dst)-> None
```

src The first source array

lower The inclusive lower boundary

upper The exclusive upper boundary

dst The destination array, must have 8u or 8s type

The function does the range check for every element of the input array:

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0$$

For single-channel arrays,

$$\texttt{dst}(I) = \texttt{lower}_0 <= \texttt{src}(I)_0 < \texttt{upper}_0 \wedge \texttt{lower}_1 <= \texttt{src}(I)_1 < \texttt{upper}_1$$

For two-channel arrays nd so forth,
'dst(I)' is set to 0xff (all 1-bits) if 'src(I)' is within the range and 0 otherwise. All the arrays must have the same size (or ROI size).

cv.InvSqrt

Calculates the inverse square root.

```
InvSqrt(value)-> float
```

value The input floating-point value

The function calculates the inverse square root of the argument, and normally it is faster than `1./sqrt(value)`. If the argument is zero or negative, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

cv.Invert

Finds the inverse or pseudo-inverse of a matrix.

```
Invert(src,dst,method=CV_LU)-> double
```

```
#define cvInv cvInvert
```

src The source matrix

dst The destination matrix

method Inversion method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix

The function inverts matrix `src1` and stores the result in `src2`.

In the case of `LU` method, the function returns the `src1` determinant (src1 must be square). If it is 0, the matrix is not inverted and `src2` is filled with zeros.

In the case of `SVD` methods, the function returns the inversed condition of `src1` (ratio of the smallest singular value to the largest singular value) and 0 if `src1` is all zeros. The SVD methods calculate a pseudo-inverse matrix if `src1` is singular.

cv.IsInf

Determines if the argument is Infinity.

```
IsInf(value)-> int
```

value The input floating-point value

The function returns 1 if the argument is $\pm\infty$ (as defined by IEEE754 standard), 0 otherwise.

cv.IsNaN

Determines if the argument is Not A Number.

```
IsNaN(value)-> int
```

value The input floating-point value

The function returns 1 if the argument is Not A Number (as defined by IEEE754 standard), 0 otherwise.

cv.LUT

Performs a look-up table transform of an array.

```
LUT(src,dst,lut)-> None
```

src Source array of 8-bit elements

dst Destination array of a given depth and of the same number of channels as the source array

lut Look-up table of 256 elements; should have the same depth as the destination array. In the case of multi-channel source and destination arrays, the table should either have a single-channel (in this case the same table is used for all channels) or the same number of channels as the source/destination array.

The function fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of `src` as follows:

$$\texttt{dst}_i \leftarrow \texttt{lut}_{\texttt{src}_i + d}$$

where

$$d = \begin{cases} 0 & \text{if } \texttt{src} \text{ has depth } \texttt{CV\_8U} \\ 128 & \text{if } \texttt{src} \text{ has depth } \texttt{CV\_8S} \end{cases}$$

---

cv.Log

Calculates the natural logarithm of every array element's absolute value.

```
Log(src,dst)-> None
```

src The source array

dst The destination array, it should have `double` type or the same type as the source

The function calculates the natural logarithm of the absolute value of every element of the input array:

$$\texttt{dst}[I] = \begin{cases} \log|\texttt{src}(I)| & \text{if } \texttt{src}[I] \neq 0 \\ \texttt{C} & \text{otherwise} \end{cases}$$

Where `C` is a large negative number (about -700 in the current implementation).

---

cv.Mahalonobis

Calculates the Mahalonobis distance between two vectors.

```
Mahalonobis(vec1,vec2,mat)-> None
```

vec1 The first 1D source vector

vec2 The second 1D source vector

mat The inverse covariance matrix

The function calculates and returns the weighted distance between two vectors:

$$d(\texttt{vec1},\texttt{vec2}) = \sqrt{\sum_{i,j} \texttt{icovar(i,j)} \cdot (\texttt{vec1}(I) - \texttt{vec2}(I)) \cdot (\texttt{vec1(j)} - \texttt{vec2(j)})}$$

The covariance matrix may be calculated using the cv.CalcCovarMatrix function and further inverted using the cv.Invert function (CV_SVD method is the prefered one because the matrix might be singular).

---

cv.Max

Finds per-element maximum of two arrays.

```
Max(src1,src2,dst)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element maximum of two arrays:

$$\texttt{dst}(I) = \max(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cv.MaxS

Finds per-element maximum of array and scalar.

```
MaxS(src,value,dst)-> None
```

src The first source array

value The scalar value

dst The destination array

The function calculates per-element maximum of array and scalar:

$$\texttt{dst}(I) = \max(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cv.Merge

Composes a multi-channel array from several single-channel arrays or inserts a single channel into the array.

```
Merge(src0,src1,src2,src3,dst)-> None
```

src0 Input channel 0

src1 Input channel 1

src2 Input channel 2

src3 Input channel 3

dst Destination array

The function is the opposite to cv.Split. If the destination array has N channels then if the first N input channels are not NULL, they all are copied to the destination array; if only a single source channel of the first N is not NULL, this particular channel is copied into the destination array; otherwise an error is raised. The rest of the source channels (beyond the first N) must always be NULL. For IplImage cv.Copy with COI set can be also used to insert a single channel into the image.

---

cv.Min

Finds per-element minimum of two arrays.

```
Min(src1,src2,dst)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

The function calculates per-element minimum of two arrays:

$$\texttt{dst}(I) = \min(\texttt{src1}(I), \texttt{src2}(I))$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cv.MinMaxLoc

Finds global minimum and maximum in array or subarray.

```
MinMaxLoc(arr,mask=NULL)-> (minVal,maxVal,minLoc,maxLoc)
```

arr The source array, single-channel or multi-channel with COI set

minVal Pointer to returned minimum value

maxVal Pointer to returned maximum value

minLoc Pointer to returned minimum location

maxLoc Pointer to returned maximum location

mask The optional mask used to select a subarray

The function finds minimum and maximum element values and their positions. The extremums are searched across the whole array, selected `ROI` (in the case of `IplImage`) or, if `mask` is not `NULL`, in the specified array region. If the array has more than one channel, it must be `IplImage` with `COI` set. In the case of multi-dimensional arrays, `minLoc->x` and `maxLoc->x` will contain raw (linear) positions of the extremums.

---

cv.MinS

Finds per-element minimum of an array and a scalar.

```
MinS(src,value,dst)-> None
```

src The first source array

value The scalar value

dst The destination array

The function calculates minimum of an array and a scalar:

$$\texttt{dst}(I) = \min(\texttt{src}(I), \texttt{value})$$

All the arrays must have a single channel, the same data type and the same size (or ROI size).

---

cv.MixChannels

Copies several channels from input arrays to certain channels of output arrays

```
MixChannels(src, dst, fromTo) -> None
```

src Input arrays

dst Destination arrays

fromTo The array of pairs of indices of the planes copied. Each pair `fromTo[k]=(i,j)` means that i-th plane from `src` is copied to the j-th plane in `dst`, where continuous plane numbering is used both in the input array list and the output array list. As a special case, when the `fromTo[k][0]` is negative, the corresponding output plane `j` is filled with zero.

The function is a generalized form of cv.cvSplit and cv.Merge and some forms of CvtColor . It can be used to change the order of the planes, add/remove alpha channel, extract or insert a single plane or multiple planes etc.

As an example, this code splits a 4-channel RGBA image into a 3-channel BGR (i.e. with R and B swapped) and separate alpha channel image:

```
rgba = cv.CreateMat(100, 100, cv.CV_8UC4)
bgr =  cv.CreateMat(100, 100, cv.CV_8UC3)
alpha = cv.CreateMat(100, 100, cv.CV_8UC1)
cv.Set(rgba, (1,2,3,4))
```

```
        cv.MixChannels([rgba], [bgr, alpha], [
           (0, 2),     # rgba[0] -> bgr[2]
           (1, 1),     # rgba[1] -> bgr[1]
           (2, 0),     # rgba[2] -> bgr[0]
           (3, 3)      # rgba[3] -> alpha[0]
        ])
```

---

cv.Mul

Calculates the per-element product of two arrays.

```
Mul(src1,src2,dst,scale)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

scale Optional scale factor

The function calculates the per-element product of two arrays:

$$\texttt{dst}(I) = \texttt{scale} \cdot \texttt{src1}(I) \cdot \texttt{src2}(I)$$

All the arrays must have the same type and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cv.MulSpectrums

Performs per-element multiplication of two Fourier spectrums.

```
MulSpectrums(src1,src2,dst,flags)-> None
```

src1 The first source array

src2 The second source array

dst The destination array of the same type and the same size as the source arrays

flags A combination of the following values;

CV_DXT_ROWS treats each row of the arrays as a separate spectrum (see cv.DFT parameters description).

CV_DXT_MUL_CONJ conjugate the second source array before the multiplication.

The function performs per-element multiplication of the two CCS-packed or complex matrices that are results of a real or complex Fourier transform.

The function, together with cv.DFT, may be used to calculate convolution of two arrays rapidly.

---

cv.MulTransposed

Calculates the product of an array and a transposed array.

```
MulTransposed(src,dst,order,delta=NULL,scale)-> None
```

src The source matrix

dst The destination matrix. Must be `CV_32F` or `CV_64F`.

order Order of multipliers

delta An optional array, subtracted from `src` before multiplication

scale An optional scaling

The function calculates the product of src and its transposition:

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})(\texttt{src} - \texttt{delta})^T$$

if $\texttt{order} = 0$, and

$$\texttt{dst} = \texttt{scale}(\texttt{src} - \texttt{delta})^T(\texttt{src} - \texttt{delta})$$

otherwise.

---

cv.Norm

Calculates absolute array norm, absolute difference norm, or relative difference norm.

```
Norm(arr1,arr2,normType=CV_L2,mask=NULL)-> double
```

arr1 The first source image

arr2 The second source image. If it is NULL, the absolute norm of `arr1` is calculated, otherwise the absolute or relative norm of `arr1`-`arr2` is calculated.

normType Type of norm, see the discussion

mask The optional operation mask

The function calculates the absolute norm of `arr1` if `arr2` is NULL:

$$norm = \begin{cases} ||\texttt{arr1}||_C = \max_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV\_C} \\ ||\texttt{arr1}||_{L1} = \sum_I |\texttt{arr1}(I)| & \text{if } \texttt{normType} = \texttt{CV\_L1} \\ ||\texttt{arr1}||_{L2} = \sqrt{\sum_I \texttt{arr1}(I)^2} & \text{if } \texttt{normType} = \texttt{CV\_L2} \end{cases}$$

or the absolute difference norm if `arr2` is not NULL:

$$norm = \begin{cases} ||\texttt{arr1} - \texttt{arr2}||_C = \max_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV\_C} \\ ||\texttt{arr1} - \texttt{arr2}||_{L1} = \sum_I |\texttt{arr1}(I) - \texttt{arr2}(I)| & \text{if } \texttt{normType} = \texttt{CV\_L1} \\ ||\texttt{arr1} - \texttt{arr2}||_{L2} = \sqrt{\sum_I (\texttt{arr1}(I) - \texttt{arr2}(I))^2} & \text{if } \texttt{normType} = \texttt{CV\_L2} \end{cases}$$

or the relative difference norm if `arr2` is not NULL and `(normType & CV_RELATIVE) != 0`:

$$norm = \begin{cases} \frac{||\texttt{arr1}-\texttt{arr2}||_C}{||\texttt{arr2}||_C} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_C} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L1}}{||\texttt{arr2}||_{L1}} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_L1} \\ \frac{||\texttt{arr1}-\texttt{arr2}||_{L2}}{||\texttt{arr2}||_{L2}} & \text{if } \texttt{normType} = \texttt{CV\_RELATIVE\_L2} \end{cases}$$

The function returns the calculated norm. A multiple-channel array is treated as a single-channel, that is, the results for all channels are combined.

---

cv.Not

Performs per-element bit-wise inversion of array elements.

```
Not(src,dst)-> None
```

src The source array

dst The destination array

The function Not inverses every bit of every array element:

```
dst(I)=~src(I)
```

---

cv.Or

Calculates per-element bit-wise disjunction of two arrays.

```
Or(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise disjunction of two arrays:

```
dst(I)=src1(I)|src2(I)
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.OrS

Calculates a per-element bit-wise disjunction of an array and a scalar.

```
OrS(src,value,dst,mask=NULL)-> None
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function OrS calculates per-element bit-wise disjunction of an array and a scalar:

```
dst(I)=src(I)|value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.PerspectiveTransform

Performs perspective matrix transformation of a vector array.

```
PerspectiveTransform(src,dst,mat)-> None
```

src The source three-channel floating-point array

dst The destination three-channel floating-point array

mat 3×3 or 4×4 transformation matrix

The function transforms every element of `src` (by treating it as 2D or 3D vector) in the following way:

$$(x, y, z) \rightarrow (x'/w, y'/w, z'/w)$$

where

$$(x', y', z', w') = \texttt{mat} \cdot \begin{bmatrix} x & y & z & 1 \end{bmatrix}$$

and

$$w = \begin{cases} w' & \text{if } w' \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

---

cv.PolarToCart

Calculates Cartesian coordinates of 2d vectors represented in polar form.

```
PolarToCart(magnitude,angle,x,y,angleInDegrees=0)-> None
```

magnitude The array of magnitudes. If it is NULL, the magnitudes are assumed to be all 1's.

angle The array of angles, whether in radians or degrees

x The destination array of x-coordinates, may be set to NULL if it is not needed

y The destination array of y-coordinates, mau be set to NULL if it is not needed

angleInDegrees The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function calculates either the x-coodinate, y-coordinate or both of every vector `magnitude(I)*exp(ang`
`j=sqrt(-1)`:

```
x(I)=magnitude(I)*cos(angle(I)),
y(I)=magnitude(I)*sin(angle(I))
```

---

cv.Pow

Raises every array element to a power.

```
Pow(src,dst,power)-> None
```

src The source array

dst The destination array, should be the same type as the source

power The exponent of power

The function raises every element of the input array to `p`:

$$\texttt{dst}[I] = \begin{cases} \texttt{src}(I)^p & \text{if } \texttt{p} \text{ is integer} \\ |\texttt{src}(I)^p| & \text{otherwise} \end{cases}$$

That is, for a non-integer power exponent the absolute values of input array elements are used. However, it is possible to get true values for negative values using some extra operations, as the following example, computing the cube root of array elements, shows:

```
CvSize size = cvGetSize(src);
CvMat* mask = cvCreateMat(size.height, size.width, CVg8UC1);
cvCmpS(src, 0, mask, CVgCMPgLT); /* find negative elements */
cvPow(src, dst, 1./3);
cvSubRS(dst, cvScalarAll(0), dst, mask); /* negate the results of negative inputs */
cvReleaseMat(&mask);
```

For some values of `power`, such as integer values, 0.5, and -0.5, specialized faster algorithms are used.

---

cv.RNG

Initializes a random number generator state.

```
RNG(seed=-1LL)-> CvRNG
```

seed 64-bit value used to initiate a random sequence

The function initializes a random number generator and returns the state. The pointer to the state can be then passed to the cv.RandInt, cv.RandReal and cv.RandArr functions. In the current implementation a multiply-with-carry generator is used.

---

cv.RandArr

Fills an array with random numbers and updates the RNG state.

```
RandArr(rng,arr,distType,param1,param2)-> None
```

rng RNG state initialized by cv.RNG

arr The destination array

distType Distribution type

- **CV_RAND_UNI** uniform distribution
- **CV_RAND_NORMAL** normal or Gaussian distribution

param1 The first parameter of the distribution. In the case of a uniform distribution it is the inclusive lower boundary of the random numbers range. In the case of a normal distribution it is the mean value of the random numbers.

param2 The second parameter of the distribution. In the case of a uniform distribution it is the exclusive upper boundary of the random numbers range. In the case of a normal distribution it is the standard deviation of the random numbers.

The function fills the destination array with uniformly or normally distributed random numbers.

---

cv.RandInt

Returns a 32-bit unsigned integer and updates RNG.

```
RandInt(rng)-> unsigned
```

rng RNG state initialized by `RandInit` and, optionally, customized by `RandSetRange` (though, the latter function does not affect the discussed function outcome)

The function returns a uniformly-distributed random 32-bit unsigned integer and updates the RNG state. It is similar to the rand() function from the C runtime library, but it always generates a 32-bit number whereas rand() returns a number in between 0 and `RAND_MAX` which is 2^{16} or 2^{32}, depending on the platform.

The function is useful for generating scalar random numbers, such as points, patch sizes, table indices, etc., where integer numbers of a certain range can be generated using a modulo operation and floating-point numbers can be generated by scaling from 0 to 1 or any other specific range.

---

cv.RandReal

Returns a floating-point random number and updates RNG.

```
RandReal(rng)-> double
```

rng RNG state initialized by cv.RNG

The function returns a uniformly-distributed random floating-point number between 0 and 1 (1 is not included).

cv.Reduce

Reduces a matrix to a vector.

```
Reduce(src,dst,dim=-1,op=CV_REDUCE_SUM)-> None
```

src The input matrix.

dst The output single-row/single-column vector that accumulates somehow all the matrix rows/-columns.

dim The dimension index along which the matrix is reduced. 0 means that the matrix is reduced to a single row, 1 means that the matrix is reduced to a single column and -1 means that the dimension is chosen automatically by analysing the dst size.

op The reduction operation. It can take of the following values:

- **CV_REDUCE_SUM** The output is the sum of all of the matrix's rows/columns.
- **CV_REDUCE_AVG** The output is the mean vector of all of the matrix's rows/columns.
- **CV_REDUCE_MAX** The output is the maximum (column/row-wise) of all of the matrix's rows/-columns.
- **CV_REDUCE_MIN** The output is the minimum (column/row-wise) of all of the matrix's rows/-columns.

The function reduces matrix to a vector by treating the matrix rows/columns as a set of 1D vectors and performing the specified operation on the vectors until a single row/column is obtained. For example, the function can be used to compute horizontal and vertical projections of an raster image. In the case of CV_REDUCE_SUM and CV_REDUCE_AVG the output may have a larger element bit-depth to preserve accuracy. And multi-channel arrays are also supported in these two reduction modes.

cv.Repeat

Fill the destination array with repeated copies of the source array.

```
Repeat(src,dst)-> None
```

src Source array, image or matrix

dst Destination array, image or matrix

The function fills the destination array with repeated copies of the source array:

```
dst(i,j)=src(i mod rows(src), j mod cols(src))
```

So the destination array may be as larger as well as smaller than the source array.

---

cv.ResetImageROI

Resets the image ROI to include the entire image and releases the ROI structure.

```
ResetImageROI(image)-> None
```

image A pointer to the image header

This produces a similar result to the following, but in addition it releases the ROI structure.

```
cvSetImageROI(image, cvRect(0, 0, image->width, image->height ));
cvSetImageCOI(image, 0);
```

---

cv.Reshape

Changes shape of matrix/image without copying data.

```
Reshape(arr, newCn, newRows=0) -> cvmat
```

arr Input array

newCn New number of channels. 'newCn = 0' means that the number of channels remains unchanged.

newRows New number of rows. 'newRows = 0' means that the number of rows remains unchanged unless it needs to be changed according to `newCn` value.

The function initializes the CvMat header so that it points to the same data as the original array but has a different shape - different number of channels, different number of rows, or both.

cv.ReshapeMatND

Changes the shape of a multi-dimensional array without copying the data.

```
ReshapeMatND(arr, newCn, newDims) -> cvmat
```

arr Input array

newDims List of new dimensions.

The function is an advanced version of cv.Reshape that can work with multi-dimensional arrays as well (though it can work with ordinary images and matrices) and change the number of dimensions.

Round

Converts a floating-point number to the nearest integer value.

```
Round(value) -> int
```

value The input floating-point value

On some architectures this function is much faster than the standard cast operations. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

Floor

Converts a floating-point number to the nearest integer value that is not larger than the argument.

```
Floor(value) -> int
```

value The input floating-point value

On some architectures this function is much faster than the standard cast operations. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

Ceil

Converts a floating-point number to the nearest integer value that is not smaller than the argument.

```
Ceil(value) -> int
```

value The input floating-point value

On some architectures this function is much faster than the standard cast operations. If the absolute value of the argument is greater than 2^{31}, the result is not determined. Special values ($\pm\infty$, NaN) are not handled.

cv.ScaleAdd

Calculates the sum of a scaled array and another array.

```
ScaleAdd(src1,scale,src2,dst)-> None
```

src1 The first source array

scale Scale factor for the first array

src2 The second source array

dst The destination array

```
#define cvMulAddS cvScaleAdd
```

The function calculates the sum of a scaled array and another array:

$$\texttt{dst}(I) = \texttt{scale}\,\texttt{src1}(I) + \texttt{src2}(I)$$

All array parameters should have the same type and the same size.

cv.Set

Sets every element of an array to a given value.

```
Set(arr,value,mask=NULL)-> None
```

arr The destination array

value Fill value

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function copies the scalar `value` to every selected element of the destination array:

$$\texttt{arr}(I) = \texttt{value} \quad \text{if} \quad \texttt{mask}(I) \neq 0$$

If array `arr` is of `IplImage` type, then is ROI used, but COI must not be set.

---

cv.Set1D

Set a specific array element.

```
Set1D(arr, idx, value) -> None
```

arr Input array

idx Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 1.

---

cv.Set2D

Set a specific array element.

```
Set2D(arr, idx0, idx1, value) -> None
```

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

value The value to assign to the element

Sets a specific array element. Array must have dimension 2.

---

cv.Set3D

Set a specific array element.

```
Set3D(arr, idx0, idx1, idx2, value) -> None
```

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 3.

---

cv.SetND

Set a specific array element.

```
SetND(arr, indices, value) -> None
```

arr Input array

indices List of zero-based element indices

value The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

cv.SetData

Assigns user data to the array header.

```
SetData(arr, data, step)-> None
```

arr Array header

data User data

step Full row length in bytes

The function assigns user data to the array header. Header should be initialized before using `cvCreate*Header`, `cvInit*Header` or cv.Mat (in the case of matrix) function.

cv.SetIdentity

Initializes a scaled identity matrix.

```
SetIdentity(mat,value=1)-> None
```

mat The matrix to initialize (not necesserily square)

value The value to assign to the diagonal elements

The function initializes a scaled identity matrix:

$$\texttt{arr}(i,j) = \begin{cases} \texttt{value} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

cv.SetImageCOI

Sets the channel of interest in an IplImage.

```
SetImageCOI(image, coi)-> None
```

image A pointer to the image header

coi The channel of interest. 0 - all channels are selected, 1 - first channel is selected, etc. Note that the channel indices become 1-based.

If the ROI is set to `NULL` and the coi is *not* 0, the ROI is allocated. Most OpenCV functions do *not* support the COI setting, so to process an individual image/matrix channel one may copy (via cv.Copy or cv.Split) the channel to a separate image/matrix, process it and then copy the result back (via cv.Copy or cv.Merge) if needed.

---

cv.SetImageROI

Sets an image Region Of Interest (ROI) for a given rectangle.

```
SetImageROI(image, rect)-> None
```

image A pointer to the image header

rect The ROI rectangle

If the original image ROI was `NULL` and the `rect` is not the whole image, the ROI structure is allocated.

Most OpenCV functions support the use of ROI and treat the image rectangle as a separate image. For example, all of the pixel coordinates are counted from the top-left (or bottom-left) corner of the ROI, not the original image.

---

cv.SetReal1D

Set a specific array element.

```
SetReal1D(arr, idx, value) -> None
```

arr Input array

idx Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 1.

cv.SetReal2D

Set a specific array element.

```
SetReal2D(arr, idx0, idx1, value) -> None
```

arr Input array

idx0 Zero-based element row index

idx1 Zero-based element column index

value The value to assign to the element

Sets a specific array element. Array must have dimension 2.

cv.SetReal3D

Set a specific array element.

```
SetReal3D(arr, idx0, idx1, idx2, value) -> None
```

arr Input array

idx0 Zero-based element index

idx1 Zero-based element index

idx2 Zero-based element index

value The value to assign to the element

Sets a specific array element. Array must have dimension 3.

cv.SetRealND

Set a specific array element.

```
SetRealND(arr, indices, value) -> None
```

arr Input array

indices List of zero-based element indices

value The value to assign to the element

Sets a specific array element. The length of array indices must be the same as the dimension of the array.

---

cv.SetZero

Clears the array.

```
SetZero(arr)-> None
```

arr Array to be cleared

The function clears the array. In the case of dense arrays (CvMat, CvMatND or IplImage), cvZero(array) is equivalent to cvSet(array,cvScalarAll(0),0). In the case of sparse arrays all the elements are removed.

---

cv.Solve

Solves a linear system or least-squares problem.

```
Solve(A,B,X,method=CV_LU)-> None
```

A The source matrix

B The right-hand part of the linear system

X The output solution

method The solution (matrix inversion) method

CV_LU Gaussian elimination with optimal pivot element chosen

CV_SVD Singular value decomposition (SVD) method

CV_SVD_SYM SVD method for a symmetric positively-defined matrix.

The function solves a linear system or least-squares problem (the latter is possible with SVD methods):

$$\texttt{dst} = argmin_X ||\texttt{src1}\,\texttt{X} - \texttt{src2}||$$

If CV_LU method is used, the function returns 1 if src1 is non-singular and 0 otherwise; in the latter case dst is not valid.

---

cv.SolveCubic

Finds the real roots of a cubic equation.

```
SolveCubic(coeffs,roots)-> None
```

coeffs The equation coefficients, an array of 3 or 4 elements

roots The output array of real roots which should have 3 elements

The function finds the real roots of a cubic equation:
If coeffs is a 4-element vector:

$$\texttt{coeffs}[0]x^3 + \texttt{coeffs}[1]x^2 + \texttt{coeffs}[2]x + \texttt{coeffs}[3] = 0$$

or if coeffs is 3-element vector:

$$x^3 + \texttt{coeffs}[0]x^2 + \texttt{coeffs}[1]x + \texttt{coeffs}[2] = 0$$

The function returns the number of real roots found. The roots are stored to root array, which is padded with zeros if there is only one root.

---

cv.Split

Divides multi-channel array into several single-channel arrays or extracts a single channel from the array.

```
Split(src,dst0,dst1,dst2,dst3)-> None
```

```
#define cvCvtPixToPlane cvSplit
```

src Source array

dst0 Destination channel 0

dst1 Destination channel 1

dst2 Destination channel 2

dst3 Destination channel 3

The function divides a multi-channel array into separate single-channel arrays. Two modes are available for the operation. If the source array has N channels then if the first N destination channels are not NULL, they all are extracted from the source array; if only a single destination channel of the first N is not NULL, this particular channel is extracted; otherwise an error is raised. The rest of the destination channels (beyond the first N) must always be NULL. For IplImage cv.Copy with COI set can be also used to extract a single channel from the image.

---

cv.Sqrt

Calculates the square root.

```
Sqrt(value)-> float
```

value The input floating-point value

The function calculates the square root of the argument. If the argument is negative, the result is not determined.

---

cv.Sub

Computes the per-element difference between two arrays.

```
Sub(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts one array from another one:

```
dst(I)=src1(I)-src2(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

---

cv.SubRS

Computes the difference between a scalar and an array.

```
SubRS(src,value,dst,mask=NULL)-> None
```

src The first source array

value Scalar to subtract from

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts every element of source array from a scalar:

```
dst(I)=value-src(I) if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cv.SubS

Computes the difference between an array and a scalar.

```
SubS(src,value,dst,mask=NULL)-> None
```

src The source array

value Subtracted scalar

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function subtracts a scalar from every element of the source array:

```
dst(I)=src(I)-value if mask(I)!=0
```

All the arrays must have the same type, except the mask, and the same size (or ROI size). For types that have limited range this operation is saturating.

cv.Sum

Adds up array elements.

```
Sum(arr)-> CvScalar
```

arr The array

The function calculates the sum `S` of array elements, independently for each channel:

$$\sum_I \texttt{arr}(I)_c$$

If the array is `IplImage` and COI is set, the function processes the selected channel only and stores the sum to the first scalar component.

cv.SVBkSb

Performs singular value back substitution.

```
SVBkSb(W,U,V,B,X,flags)-> None
```

W Matrix or vector of singular values

U Left orthogonal matrix (tranposed, perhaps)

V Right orthogonal matrix (tranposed, perhaps)

B The matrix to multiply the pseudo-inverse of the original matrix A by. This is an optional parameter. If it is omitted then it is assumed to be an identity matrix of an appropriate size (so that X will be the reconstructed pseudo-inverse of A).

X The destination matrix: result of back substitution

flags Operation flags, should match exactly to the flags passed to cv.SVD

The function calculates back substitution for decomposed matrix A (see cv.SVD description) and matrix B:

$$\mathrm{X} = \mathrm{V}\mathrm{W}^{-1}\mathrm{U}^{T}\mathrm{B}$$

where

$$W^{-1}_{(i,i)} = \begin{cases} 1/W_{(i,i)} & \text{if } W_{(i,i)} > \epsilon \sum_i W_{(i,i)} \\ 0 & \text{otherwise} \end{cases}$$

and ϵ is a small number that depends on the matrix data type.

This function together with cv.SVD is used inside cv.Invert and cv.Solve, and the possible reason to use these (svd and bksb) "low-level" function, is to avoid allocation of temporary matrices inside the high-level counterparts (inv and solve).

cv.SVD

Performs singular value decomposition of a real floating-point matrix.

```
SVD(A,W, U = None, V = None, flags=0)-> None
```

A Source $\mathtt{M} \times \mathtt{N}$ matrix

W Resulting singular value diagonal matrix ($\mathtt{M} \times \mathtt{N}$ or $\min(\mathtt{M}, \mathtt{N}) \times \min(\mathtt{M}, \mathtt{N})$) or $\min(\mathtt{M}, \mathtt{N}) \times 1$ vector of the singular values

U Optional left orthogonal matrix, $\mathtt{M} \times \min(\mathtt{M}, \mathtt{N})$ (when CV_SVD_U_T is not set), or $\min(\mathtt{M}, \mathtt{N}) \times \mathtt{M}$ (when CV_SVD_U_T is set), or $\mathtt{M} \times \mathtt{M}$ (regardless of CV_SVD_U_T flag).

V Optional right orthogonal matrix, $\mathtt{N} \times \min(\mathtt{M}, \mathtt{N})$ (when CV_SVD_V_T is not set), or $\min(\mathtt{M}, \mathtt{N}) \times \mathtt{N}$ (when CV_SVD_V_T is set), or $\mathtt{N} \times \mathtt{N}$ (regardless of CV_SVD_V_T flag).

flags Operation flags; can be 0 or a combination of the following values:

CV_SVD_MODIFY_A enables modification of matrix A during the operation. It speeds up the processing.

CV_SVD_U_T means that the transposed matrix U is returned. Specifying the flag speeds up the processing.

CV_SVD_V_T means that the transposed matrix V is returned. Specifying the flag speeds up the processing.

The function decomposes matrix A into the product of a diagonal matrix and two orthogonal matrices:

$$A = U\, W\, V^T$$

where W is a diagonal matrix of singular values that can be coded as a 1D vector of singular values and U and V. All the singular values are non-negative and sorted (together with U and V columns) in descending order.

An SVD algorithm is numerically robust and its typical applications include:

- accurate eigenvalue problem solution when matrix A is a square, symmetric, and positively defined matrix, for example, when it is a covariance matrix. W in this case will be a vector/-matrix of the eigenvalues, and $U = V$ will be a matrix of the eigenvectors.
- accurate solution of a poor-conditioned linear system.
- least-squares solution of an overdetermined linear system. This and the preceeding is done by using the cv.Solve function with the CV_SVD method.

- accurate calculation of different matrix characteristics such as the matrix rank (the number of non-zero singular values), condition number (ratio of the largest singular value to the smallest one), and determinant (absolute value of the determinant is equal to the product of singular values).

---

cv.Trace

Returns the trace of a matrix.

```
Trace(mat)-> CvScalar
```

mat The source matrix

The function returns the sum of the diagonal elements of the matrix `src1`.

$$tr(\texttt{mat}) = \sum_i \texttt{mat}(i,i)$$

---

cv.Transform

Performs matrix transformation of every array element.

```
Transform(src,dst,transmat,shiftvec=NULL)-> None
```

src The first source array

dst The destination array

transmat Transformation matrix

shiftvec Optional shift vector

The function performs matrix transformation of every element of array `src` and stores the results in `dst`:

$$dst(I) = transmat \cdot src(I) + shiftvec$$

That is, every element of an N-channel array src is considered as an N-element vector which is transformed using a M × N matrix transmat and shift vector shiftvec into an element of M-channel array dst. There is an option to embedd shiftvec into transmat. In this case transmat should be a M × $(N + 1)$ matrix and the rightmost column is treated as the shift vector.

Both source and destination arrays should have the same depth and the same size or selected ROI size. transmat and shiftvec should be real floating-point matrices.

The function may be used for geometrical transformation of n dimensional point set, arbitrary linear color space transformation, shuffling the channels and so forth.

---

cv.Transpose

Transposes a matrix.

```
Transpose(src,dst)-> None
```

```
#define cvT cvTranspose
```

src The source matrix

dst The destination matrix

The function transposes matrix src1:

$$\texttt{dst}(i,j) = \texttt{src}(j,i)$$

Note that no complex conjugation is done in the case of a complex matrix. Conjugation should be done separately: look at the sample code in cv.XorS for an example.

---

cv.Xor

Performs per-element bit-wise ”exclusive or” operation on two arrays.

```
Xor(src1,src2,dst,mask=NULL)-> None
```

src1 The first source array

src2 The second source array

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function calculates per-element bit-wise logical conjunction of two arrays:

```
dst(I)=src1(I)^src2(I) if mask(I)!=0
```

In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size.

---

cv.XorS

Performs per-element bit-wise ”exclusive or” operation on an array and a scalar.

```
XorS(src,value,dst,mask=NULL)-> None
```

src The source array

value Scalar to use in the operation

dst The destination array

mask Operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The function XorS calculates per-element bit-wise conjunction of an array and a scalar:

```
dst(I)=src(I)^value if mask(I)!=0
```

Prior to the actual operation, the scalar is converted to the same type as that of the array(s). In the case of floating-point arrays their bit representations are used for the operation. All the arrays must have the same type, except the mask, and the same size

The following sample demonstrates how to conjugate complex vector by switching the most-significant bit of imaging part:

```
float a[] = { 1, 0, 0, 1, -1, 0, 0, -1 }; /* 1, j, -1, -j */
CvMat A = cvMat(4, 1, CV\_32FC2, &a);
int i, negMask = 0x80000000;
cvXorS(&A, cvScalar(0, *(float*)&negMask, 0, 0 ), &A, 0);
for(i = 0; i < 4; i++ )
    printf("(%.1f, %.1f) ", a[i*2], a[i*2+1]);
```

The code should print:

```
(1.0,0.0) (0.0,-1.0) (-1.0,0.0) (0.0,1.0)
```

---

cv.mGet

Returns the particular element of single-channel floating-point matrix.

```
mGet(mat,row,col)-> double
```

mat Input matrix

row The zero-based index of row

col The zero-based index of column

The function is a fast replacement for cv.GetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

---

cv.mSet

Returns a specific element of a single-channel floating-point matrix.

```
mSet(mat,row,col,value)-> None
```

mat The matrix

row The zero-based index of row

col The zero-based index of column

value The new value of the matrix element

The function is a fast replacement for cv.SetReal2D in the case of single-channel floating-point matrices. It is faster because it is inline, it does fewer checks for array type and array element type, and it checks for the row and column ranges only in debug mode.

12.3 Dynamic Structures

CvSeq

Growable sequence of elements.

Many OpenCV functions return a CvSeq object. The CvSeq obect is a sequence, so these are all legal:

```
seq = cv.FindContours(scribble, storage, cv.CV_RETR_CCOMP, cv.CV_CHAIN_APPROX_SIMPLE)
# seq is a sequence of point pairs
print len(seq)
# FindContours returns a sequence of (x,y) points, so to print them out:
for (x,y) in seq:
   print (x,y)
print seq[10]             # tenth entry in the seqeuence
print seq[::-1]           # reversed sequence
print sorted(list(seq))   # sorted sequence
```

Also, a CvSeq object has methods `h_next()`, `h_prev()`, `v_next()` and `v_prev()`. Some OpenCV functions (for example cv.FindContours) can return multiple CvSeq objects, connected by these relations. In this case the methods return the other sequences. If no relation between sequences exists, then the methods return `None`.

CvSet

Collection of nodes.

Some OpenCV functions return a CvSet object. The CvSet obect is iterable, for example:

```
for i in s:
  print i
print set(s)
print list(s)
```

cv.CloneSeq

Creates a copy of a sequence.

```
CloneSeq(seq,storage)-> None
```

seq Sequence

storage The destination storage block to hold the new sequence header and the copied data, if any. If it is NULL, the function uses the storage block containing the input sequence.

The function makes a complete copy of the input sequence and returns it.

---

cv.CreateMemStorage

Creates memory storage.

```
CreateMemStorage(blockSize = 0) -> memstorage
```

blockSize Size of the storage blocks in bytes. If it is 0, the block size is set to a default value - currently it is about 64K.

The function creates an empty memory storage. See CvMemStorage description.

---

cv.SeqInvert

Reverses the order of sequence elements.

```
SeqInvert(seq)-> None
```

seq Sequence

The function reverses the sequence in-place - makes the first element go last, the last element go first and so forth.

---

cv.SeqRemove

Removes an element from the middle of a sequence.

```
SeqRemove(seq,index)-> None
```

seq Sequence

index Index of removed element

The function removes elements with the given index. If the index is out of range the function reports an error. An attempt to remove an element from an empty sequence is a special case of this situation. The function removes an element by shifting the sequence elements between the nearest end of the sequence and the `index`-th position, not counting the latter.

---

cv.SeqRemoveSlice

Removes a sequence slice.

```
SeqRemoveSlice(seq,slice)-> None
```

seq Sequence

slice The part of the sequence to remove

The function removes a slice from the sequence.

12.4 Drawing Functions

Drawing functions work with matrices/images of arbitrary depth. The boundaries of the shapes can be rendered with antialiasing (implemented only for 8-bit images for now). All the functions include the parameter color that uses a rgb value (that may be constructed with `CV_RGB`) for color images and brightness for grayscale images. For color images the order channel is normally *Blue, Green, Red*, this is what cv., cv. and cv. expect

If you are using your own image rendering and I/O functions, you can use any channel ordering, the drawing functions process each channel independently and do not depend on the channel order or even on the color space used. The whole image can be converted from BGR to RGB or to a different color space using cv..

If a drawn figure is partially or completely outside the image, the drawing functions clip it. Also, many drawing functions can handle pixel coordinates specified with sub-pixel accuracy, that is, the coordinates can be passed as fixed-point numbers, encoded as integers. The number of fractional bits is specified by the `shift` parameter and the real point coordinates are calculated as $\texttt{Point}(x, y) \rightarrow \texttt{Point2f}(x*2^{-shift}, y*2^{-shift})$. This feature is especially effective wehn rendering antialiased shapes.

Also, note that the functions do not support alpha-transparency - when the target image is 4-channnel, then the `color[3]` is simply copied to the repainted pixels. Thus, if you want to paint semi-transparent shapes, you can paint them in a separate buffer and then blend it with the main image.

cv.Circle

Draws a circle.

```
Circle(img,center,radius,color,thickness=1,lineType=8,shift=0)-> None
```

img Image where the circle is drawn

center Center of the circle

radius Radius of the circle

color Circle color

thickness Thickness of the circle outline if positive, otherwise this indicates that a filled circle is to be drawn

lineType Type of the circle boundary, see Line description

shift Number of fractional bits in the center coordinates and radius value

The function draws a simple or filled circle with a given center and radius.

cv.ClipLine

Clips the line against the image rectangle.

```
ClipLine(imgSize, pt1, pt2) -> (clipped_pt1, clipped_pt2)
```

imgSize Size of the image

pt1 First ending point of the line segment.

pt2 Second ending point of the line segment.

The function calculates a part of the line segment which is entirely within the image. If the line segment is outside the image, it returns None. If the line segment is inside the image it returns a new pair of points.

cv.DrawContours

Draws contour outlines or interiors in an image.

```
DrawContours(img,contour,external_color,hole_color,max_level,thickness=1,lineType=8,
None
```

img Image where the contours are to be drawn. As with any other drawing function, the contours are clipped with the ROI.

contour Pointer to the first contour

external_color Color of the external contours

hole_color Color of internal contours (holes)

max_level Maximal level for drawn contours. If 0, only `contour` is drawn. If 1, the contour and all contours following it on the same level are drawn. If 2, all contours following and all contours one level below the contours are drawn, and so forth. If the value is negative, the function does not draw the contours following after `contour` but draws the child contours of `contour` up to the $|\texttt{max\_level}| - 1$ level.

thickness Thickness of lines the contours are drawn with. If it is negative (For example, =CV_FILLED), the contour interiors are drawn.

lineType Type of the contour segments, see Line description

The function draws contour outlines in the image if $\texttt{thickness} \geq 0$ or fills the area bounded by the contours if $\texttt{thickness} < 0$.

cv.Ellipse

Draws a simple or thick elliptic arc or an fills ellipse sector.

```
Ellipse(img,center,axes,angle,start_angle,end_angle,color,thickness=1,lineType=8,shi
None
```

img The image

center Center of the ellipse

axes Length of the ellipse axes

angle Rotation angle

start_angle Starting angle of the elliptic arc

end_angle Ending angle of the elliptic arc.

color Ellipse color

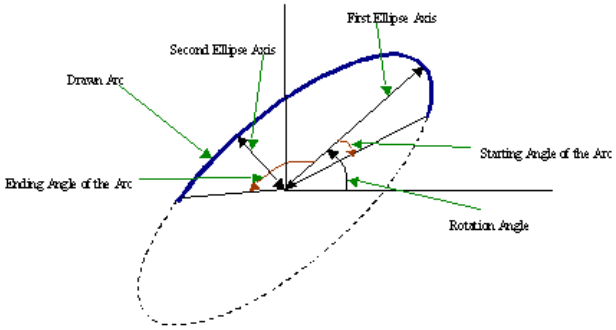
thickness Thickness of the ellipse arc outline if positive, otherwise this indicates that a filled ellipse sector is to be drawn

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the center coordinates and axes' values

The function draws a simple or thick elliptic arc or fills an ellipse sector. The arc is clipped by the ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs. All the angles are given in degrees. The picture below explains the meaning of the parameters.

Parameters of Elliptic Arc



---

cv.EllipseBox

Draws a simple or thick elliptic arc or fills an ellipse sector.

```
EllipseBox(img,box,color,thickness=1,lineType=8,shift=0)-> None
```

img Image

box The enclosing box of the ellipse drawn

thickness Thickness of the ellipse boundary

lineType Type of the ellipse boundary, see Line description

shift Number of fractional bits in the box vertex coordinates

The function draws a simple or thick ellipse outline, or fills an ellipse. The functions provides a convenient way to draw an ellipse approximating some shape; that is what CamShift and FitEllipse do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

---

cv.FillConvexPoly

Fills a convex polygon.

```
FillConvexPoly(img,pn,color,lineType=8,shift=0)-> None
```

img Image

pn List of coordinate pairs

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills a convex polygon's interior. This function is much faster than the function `cvFillPoly` and can fill not only convex polygons but any monotonic polygon, i.e., a polygon whose contour intersects every horizontal line (scan line) twice at the most.

---

cv.FillPoly

Fills a polygon's interior.

```
FillPoly(img,polys,color,lineType=8,shift=0)-> None
```

img Image

polys List of lists of (x,y) pairs. Each list of points is a polygon.

color Polygon color

lineType Type of the polygon boundaries, see Line description

shift Number of fractional bits in the vertex coordinates

The function fills an area bounded by several polygonal contours. The function fills complex areas, for example, areas with holes, contour self-intersection, and so forth.

---

cv.GetTextSize

Retrieves the width and height of a text string.

```
GetTextSize(textString,font)-> (textSize,baseline)
```

font Pointer to the font structure

textString Input string

textSize Resultant size of the text string. Height of the text does not include the height of character parts that are below the baseline.

baseline y-coordinate of the baseline relative to the bottom-most text point

The function calculates the dimensions of a rectangle to enclose a text string when a specified font is used.

---

cv.InitFont

Initializes font structure.

```
InitFont(fontFace,hscale,vscale,shear=0,thickness=1,lineType=8)-> font
```

font Pointer to the font structure initialized by the function

fontFace Font name identifier. Only a subset of Hershey fonts http://sources.isc.org/utils/misc/hershey-font.txt are supported now:

- **CV_FONT_HERSHEY_SIMPLEX** normal size sans-serif font
- **CV_FONT_HERSHEY_PLAIN** small size sans-serif font
- **CV_FONT_HERSHEY_DUPLEX** normal size sans-serif font (more complex than CV_FONT_HERSHEY_SIMPLEX)
- **CV_FONT_HERSHEY_COMPLEX** normal size serif font
- **CV_FONT_HERSHEY_TRIPLEX** normal size serif font (more complex than CV_FONT_HERSHEY_COMPLEX)
- **CV_FONT_HERSHEY_COMPLEX_SMALL** smaller version of CV_FONT_HERSHEY_COMPLEX
- **CV_FONT_HERSHEY_SCRIPT_SIMPLEX** hand-writing style font
- **CV_FONT_HERSHEY_SCRIPT_COMPLEX** more complex variant of CV_FONT_HERSHEY_SCRIPT_SIMPLEX

The parameter can be composited from one of the values above and an optional CV_FONT_ITALIC flag, which indicates italic or oblique font.

hscale Horizontal scale. If equal to 1.0f, the characters have the original width depending on the font type. If equal to 0.5f, the characters are of half the original width.

vscale Vertical scale. If equal to 1.0f, the characters have the original height depending on the font type. If equal to 0.5f, the characters are of half the original height.

shear Approximate tangent of the character slope relative to the vertical line. A zero value means a non-italic font, 1.0f means about a 45 degree slope, etc.

thickness Thickness of the text strokes

lineType Type of the strokes, see Line description

The function initializes the font structure that can be passed to text rendering functions.

---

cv.InitLineIterator

Initializes the line iterator.

```
InitLineIterator(image, pt1, pt2, connectivity=8, left_to_right=0) ->
None
```

image Image to sample the line from

pt1 First ending point of the line segment

pt2 Second ending point of the line segment

connectivity The scanned line connectivity, 4 or 8.

left_to_right If ($\texttt{left\_to\_right} = 0$) then the line is scanned in the specified order, from `pt1` to `pt2`. If ($\texttt{left\_to\_right} \neq 0$) the line is scanned from left-most point to right-most.

The function initializes the line iterator and returns the number of pixels between the two end points. Both points must be inside the image. After the iterator has been initialized, all the points on the raster line that connects the two ending points may be retrieved by successive calls of `CV_NEXT_LINE_POINT` point. The points on the line are calculated one by one using a 4-connected or 8-connected Bresenham algorithm.

---

cv.Line

Draws a line segment connecting two points.

```
Line(img,pt1,pt2,color,thickness=1,lineType=8,shift=0)-> None
```

img The image

pt1 First point of the line segment

pt2 Second point of the line segment

color Line color

thickness Line thickness

lineType Type of the line:

- **8** (or omitted) 8-connected line.
- **4** 4-connected line.
- **CV_AA** antialiased line.

shift Number of fractional bits in the point coordinates

The function draws the line segment between `pt1` and `pt2` points in the image. The line is clipped by the image or ROI rectangle. For non-antialiased lines with integer coordinates the 8-connected or 4-connected Bresenham algorithm is used. Thick lines are drawn with rounding endings. Antialiased lines are drawn using Gaussian filtering. To specify the line color, the user may use the macro `CV_RGB( r, g, b )`.

cv.PolyLine

Draws simple or thick polygons.

```
PolyLine(img,polys,is_closed,color,thickness=1,lineType=8,shift=0)->
None
```

polys List of lists of (x,y) pairs. Each list of points is a polygon.

img Image

is_closed Indicates whether the polylines must be drawn closed. If closed, the function draws the line from the last vertex of every contour to the first vertex.

color Polyline color

thickness Thickness of the polyline edges

lineType Type of the line segments, see Line description

shift Number of fractional bits in the vertex coordinates

The function draws single or multiple polygonal curves.

cv.PutText

Draws a text string.

```
PutText(img,text,org,font,color)-> None
```

img Input image

text String to print

org Coordinates of the bottom-left corner of the first letter

font Pointer to the font structure

color Text color

The function renders the text in the image with the specified font and color. The printed text is clipped by the ROI rectangle. Symbols that do not belong to the specified font are replaced with the symbol for a rectangle.

cv.Rectangle

Draws a simple, thick, or filled rectangle.

```
Rectangle(img,pt1,pt2,color,thickness=1,lineType=8,shift=0)-> None
```

img Image

pt1 One of the rectangle's vertices

pt2 Opposite rectangle vertex

color Line color (RGB) or brightness (grayscale image)

thickness Thickness of lines that make up the rectangle. Negative values, e.g., CV_FILLED, cause the function to draw a filled rectangle.

lineType Type of the line, see Line description

shift Number of fractional bits in the point coordinates

The function draws a rectangle with two opposite corners `pt1` and `pt2`.

CV_RGB

Constructs a color value.

```
CV_RGB(red,grn,blu)->CvScalar
```

red Red component

grn Green component

blu Blue component

12.5 XML/YAML Persistence

cv.Load

Loads an object from a file.

```
Load(filename,storage=NULL,name=NULL)-> generic
```

filename File name

storage Memory storage for dynamic structures, such as CvSeq or CvGraph . It is not used for matrices or images.

name Optional object name. If it is NULL, the first top-level object in the storage will be loaded.

The function loads an object from a file. It provides a simple interface to cv.Read. After the object is loaded, the file storage is closed and all the temporary buffers are deleted. Thus, to load a dynamic structure, such as a sequence, contour, or graph, one should pass a valid memory storage destination to the function.

cv.Save

Saves an object to a file.

```
Save(filename,structPtr,name=NULL,comment=NULL)-> None
```

filename File name

structPtr Object to save

name Optional object name. If it is NULL, the name will be formed from `filename`.

comment Optional comment to put in the beginning of the file

The function saves an object to a file. It provides a simple interface to Write .

12.6 Clustering and Search in Multi-Dimensional Spaces

cv.KMeans2

Splits set of vectors by a given number of clusters.

```
KMeans2(samples,nclusters,labels,termcrit)-> None
```

samples Floating-point matrix of input samples, one row per sample

nclusters Number of clusters to split the set by

labels Output integer vector storing cluster indices for every sample

termcrit Specifies maximum number of iterations and/or accuracy (distance the centers can move by between subsequent iterations)

The function `cvKMeans2` implements a k-means algorithm that finds the centers of `nclusters` clusters and groups the input samples around the clusters. On output, $\texttt{labels}_i$ contains a cluster index for samples stored in the i-th row of the `samples` matrix.

12.7 Utility and System Functions and Macros

Error Handling

Errors in argument type cause a `TypeError` exception. OpenCV errors cause an cv.error exception. For example:

```
>>> import cv
>>> cv.LoadImage(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: argument 1 must be string, not int
>>> cv.CreateMat(-1, -1, cv.CV_8UC1)
OpenCV Error: Incorrect size of input array (Non-positive width or height) in cvCreateMatHe
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
cv.error: Non-positive width or height
```

cv.GetTickCount

Returns the number of ticks.

```
GetTickCount() -> long
```

The function returns number of the ticks starting from some platform-dependent event (number of CPU ticks from the startup, number of milliseconds from 1970th year, etc.). The function is useful for accurate measurement of a function/user-code execution time. To convert the number of ticks to time units, use cv.GetTickFrequency.

cv.GetTickFrequency

Returns the number of ticks per microsecond.

```
GetTickFrequency() -> long
```

The function returns the number of ticks per microsecond. Thus, the quotient of cv.GetTickCount and cv.GetTickFrequency will give the number of microseconds starting from the platform-dependent event.

Chapter 13

cv. Image Processing and Computer Vision

13.1 Image Filtering

Functions and classes described in this section are used to perform various linear or non-linear filtering operations on 2D images (represented as cv.'s), that is, for each pixel location (x, y) in the source image some its (normally rectangular) neighborhood is considered and used to compute the response. In case of a linear filter it is a weighted sum of pixel values, in case of morphological operations it is the minimum or maximum etc. The computed response is stored to the destination image at the same location (x, y). It means, that the output image will be of the same size as the input image. Normally, the functions supports multi-channel arrays, in which case every channel is processed independently, therefore the output image will also have the same number of channels as the input one.

Another common feature of the functions and classes described in this section is that, unlike simple arithmetic functions, they need to extrapolate values of some non-existing pixels. For example, if we want to smooth an image using a Gaussian 3×3 filter, then during the processing of the left-most pixels in each row we need pixels to the left of them, i.e. outside of the image. We can let those pixels be the same as the left-most image pixels (i.e. use "replicated border" extrapolation method), or assume that all the non-existing pixels are zeros ("contant border" extrapolation method) etc.

IplConvKernel

An IplConvKernel is a rectangular convolution kernel, created by function CreateStructuringElementEx .

cv.CopyMakeBorder

Copies an image and makes a border around it.

```
CopyMakeBorder(src,dst,offset,bordertype,value=(0,0,0,0))-> None
```

src The source image

dst The destination image

offset Coordinates of the top-left corner (or bottom-left in the case of images with bottom-left origin) of the destination image rectangle where the source image (or its ROI) is copied. Size of the rectanlge matches the source image size/ROI size

bordertype Type of the border to create around the copied source image rectangle; types include:

IPL_BORDER_CONSTANT border is filled with the fixed value, passed as last parameter of the function.

IPL_BORDER_REPLICATE the pixels from the top and bottom rows, the left-most and right-most columns are replicated to fill the border.

(The other two border types from IPL, `IPL_BORDER_REFLECT` and `IPL_BORDER_WRAP`, are currently unsupported)

value Value of the border pixels if `bordertype` is `IPL_BORDER_CONSTANT`

The function copies the source 2D array into the interior of the destination array and makes a border of the specified type around the copied area. The function is useful when one needs to emulate border type that is different from the one embedded into a specific algorithm implementation. For example, morphological functions, as well as most of other filtering functions in OpenCV, internally use replication border type, while the user may need a zero border or a border, filled with 1's or 255's.

cv.CreateStructuringElementEx

Creates a structuring element.

```
CreateStructuringElementEx(cols,rows,anchorX,anchorY,shape,values=None)->
kernel
```

cols Number of columns in the structuring element

rows Number of rows in the structuring element

anchorX Relative horizontal offset of the anchor point

anchorY Relative vertical offset of the anchor point

shape Shape of the structuring element; may have the following values:

CV_SHAPE_RECT a rectangular element

CV_SHAPE_CROSS a cross-shaped element

CV_SHAPE_ELLIPSE an elliptic element

CV_SHAPE_CUSTOM a user-defined element. In this case the parameter `values` specifies the mask, that is, which neighbors of the pixel must be considered

values Pointer to the structuring element data, a plane array, representing row-by-row scanning of the element matrix. Non-zero values indicate points that belong to the element. If the pointer is `NULL`, then all values are considered non-zero, that is, the element is of a rectangular shape. This parameter is considered only if the shape is `CV_SHAPE_CUSTOM`

The function CreateStructuringElementEx allocates and fills the structure `IplConvKernel`, which can be used as a structuring element in the morphological operations.

---

cv.Dilate

Dilates an image by using a specific structuring element.

```
Dilate(src,dst,element=NULL,iterations=1)-> None
```

src Source image

dst Destination image

element Structuring element used for dilation. If it is NULL, a 3×3 rectangular structuring element is used

iterations Number of times dilation is applied

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\max_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Dilation can be applied several (iterations) times. For color images, each channel is processed independently.

---

cv.Erode

Erodes an image by using a specific structuring element.

```
Erode(src,dst,element=NULL,iterations=1)-> None
```

src Source image

dst Destination image

element Structuring element used for erosion. If it is NULL, a 3×3 rectangular structuring element is used

iterations Number of times erosion is applied

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\min_{(x',y') \, in \, \texttt{element}} src(x+x', y+y')$$

The function supports the in-place mode. Erosion can be applied several (iterations) times. For color images, each channel is processed independently.

---

cv.Filter2D

Convolves an image with the kernel.

```
Filter2D(src,dst,kernel,anchor=(-1,-1))-> None
```

src The source image

dst The destination image

kernel Convolution kernel, a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cv.Split and process them individually

anchor The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor shoud lie within the kernel. The special default value (-1,-1) means that it is at the kernel center

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values from the nearest pixels that are inside the image.

---

cv.Laplace

Calculates the Laplacian of an image.

```
Laplace(src,dst,apertureSize=3)-> None
```

src Source image

dst Destination image

apertureSize Aperture size (it has the same meaning as cv.Sobel)

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\texttt{dst}(x,y) = \frac{d^2\texttt{src}}{dx^2} + \frac{d^2\texttt{src}}{dy^2}$$

Setting `apertureSize` = 1 gives the fastest variant that is equal to convolving the image with the following kernel:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Similar to the cv.Sobel function, no scaling is done and the same combinations of input and output formats are supported.

---

cv.MorphologyEx

Performs advanced morphological transformations.

```
MorphologyEx(src,dst,temp,element,operation,iterations=1)-> None
```

src Source image

dst Destination image

temp Temporary image, required in some cases

element Structuring element

operation Type of morphological operation, one of the following:

- **CV_MOP_OPEN** opening
- **CV_MOP_CLOSE** closing
- **CV_MOP_GRADIENT** morphological gradient
- **CV_MOP_TOPHAT** "top hat"
- **CV_MOP_BLACKHAT** "black hat"

iterations Number of times erosion and dilation are applied

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$dst = open(src, element) = dilate(erode(src, element), element)$$

Closing:

$$dst = close(src, element) = erode(dilate(src, element), element)$$

Morphological gradient:

$$dst = morph\_grad(src, element) = dilate(src, element) - erode(src, element)$$

"Top hat":

$$dst = tophat(src, element) = src - open(src, element)$$

"Black hat":

$$dst = blackhat(src, element) = close(src, element) - src$$

The temporary image `temp` is required for a morphological gradient and, in the case of in-place operation, for "top hat" and "black hat".

---

cv.PyrDown

Downsamples an image.

```
PyrDown(src,dst,filter=CV_GAUSSIAN_5X5)-> None
```

src The source image

dst The destination image, should have a half as large width and height than the source

filter Type of the filter used for convolution; only `CV_GAUSSIAN_5x5` is currently supported

The function performs the downsampling step of the Gaussian pyramid decomposition. First it convolves the source image with the specified filter and then downsamples the image by rejecting even rows and columns.

---

cv.Smooth

Smooths the image in one of several ways.

```
Smooth(src,dst,smoothtype=CV_GAUSSIAN,param1=3,param2=0,param3=0,param4=0)->
None
```

src The source image

dst The destination image

smoothtype Type of the smoothing:

- **CV_BLUR_NO_SCALE** linear convolution with `param1` × `param2` box kernel (all 1's). If you want to smooth different pixels with different-size box kernels, you can use the integral image that is computed using cv.Integral
- **CV_BLUR** linear convolution with `param1` × `param2` box kernel (all 1's) with subsequent scaling by $1/(\texttt{param1} \cdot \texttt{param2})$
- **CV_GAUSSIAN** linear convolution with a `param1` × `param2` Gaussian kernel
- **CV_MEDIAN** median filter with a `param1` × `param1` square aperture
- **CV_BILATERAL** bilateral filter with a `param1`×`param1` square aperture, color sigma=`param3` and spatial sigma=`param4`. If `param1=0`, the aperture square side is set to `cvRound(param4*1.5)*` Information about bilateral filtering can be found at http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1 The first parameter of the smoothing operation, the aperture width. Must be a positive odd number (1, 3, 5, ...)

param2 The second parameter of the smoothing operation, the aperture height. Ignored by `CV_MEDIAN` and `CV_BILATERAL` methods. In the case of simple scaled/non-scaled and Gaussian blur if `param2` is zero, it is set to `param1`. Otherwise it must be a positive odd number.

param3 In the case of a Gaussian parameter this parameter may specify Gaussian σ (standard deviation). If it is zero, it is calculated from the kernel size:

$$\sigma = 0.3(n/2 - 1) + 0.8 \quad \text{where} \quad n = \begin{array}{l} \texttt{param1} \text{ for horizontal kernel} \\ \texttt{param2} \text{ for vertical kernel} \end{array}$$

Using standard sigma for small kernels (3×3 to 7×7) gives better speed. If `param3` is not zero, while `param1` and `param2` are zeros, the kernel size is calculated from the sigma (to provide accurate enough operation).

The function smooths an image using one of several methods. Every of the methods has some features and restrictions listed below

Blur with no scaling works with single-channel images only and supports accumulation of 8-bit to 16-bit format (similar to cv.Sobel and cv.Laplace) and 32-bit floating point to 32-bit floating-point format.

Simple blur and Gaussian blur support 1- or 3-channel, 8-bit and 32-bit floating point images. These two methods can process images in-place.

Median and bilateral filters work with 1- or 3-channel 8-bit images and can not process images in-place.

cv.Sobel

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator.

```
Sobel(src,dst,xorder,yorder,apertureSize = 3)-> None
```

src Source image of type CvArr*

dst Destination image

xorder Order of the derivative x

yorder Order of the derivative y

apertureSize Size of the extended Sobel kernel, must be 1, 3, 5 or 7

In all cases except 1, an `apertureSize` $\times$ `apertureSize` separable kernel will be used to calculate the derivative. For `apertureSize` $= 1$ a 3×1 or 1×3 a kernel is used (Gaussian smoothing is not done). There is also the special value `CV_SCHARR` (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\texttt{dst}(x,y) = \frac{d^{xorder+yorder}\texttt{src}}{dx^{xorder} \cdot dy^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation so the result is more or less resistant to the noise. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `apertureSize` = 3) or (`xorder` = 0, `yorder` = 1, `apertureSize` = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

and the second one corresponds to a kernel of:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

or a kernel of:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix}$$

depending on the image origin (`origin` field of `IplImage` structure). No scaling is done, so the destination image usually has larger numbers (in absolute values) than the source image does. To avoid overflow, the function requires a 16-bit destination image if the source image is 8-bit. The result can be converted back to 8-bit using the cv.ConvertScale or the cv.ConvertScaleAbs function. Besides 8-bit images the function can process 32-bit floating-point images. Both the source and the destination must be single-channel images of equal size or equal ROI size.

13.2 Geometric Image Transformations

The functions in this section perform various geometrical transformations of 2D images. That is, they do not change the image content, but deform the pixel grid, and map this deformed grid to the destination image. In fact, to avoid sampling artifacts, the mapping is done in the reverse order, from destination to the source. That is, for each pixel (x, y) of the destination image, the functions compute coordinates of the corresponding "donor" pixel in the source image and copy the pixel value, that is:

$$\texttt{dst}(x, y) = \texttt{src}(f_x(x, y), f_y(x, y))$$

In the case when the user specifies the forward mapping: $\langle g_x, g_y \rangle : \texttt{src} \rightarrow \texttt{dst}$, the OpenCV functions first compute the corresponding inverse mapping: $\langle f_x, f_y \rangle : \texttt{dst} \rightarrow \texttt{src}$ and then use the above formula.

The actual implementations of the geometrical transformations, from the most generic cv.Remap and to the simplest and the fastest cv.Resize, need to solve the 2 main problems with the above formula:

1. extrapolation of non-existing pixels. Similarly to the filtering functions, described in the previous section, for some (x, y) one of $f_x(x, y)$ or $f_y(x, y)$, or they both, may fall outside of the image, in which case some extrapolation method needs to be used. OpenCV provides the same selection of the extrapolation methods as in the filtering functions, but also an additional method `BORDER_TRANSPARENT`, which means that the corresponding pixels in the destination image will not be modified at all.

2. interpolation of pixel values. Usually $f_x(x,y)$ and $f_y(x,y)$ are floating-point numbers (i.e. $\langle f_x, f_y \rangle$ can be an affine or perspective transformation, or radial lens distortion correction etc.), so a pixel values at fractional coordinates needs to be retrieved. In the simplest case the coordinates can be just rounded to the nearest integer coordinates and the corresponding pixel used, which is called nearest-neighbor interpolation. However, a better result can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel $(f_x(x,y), f_y(x,y))$ and then the value of the polynomial at $(f_x(x,y), f_y(x,y))$ is taken as the interpolated pixel value. In OpenCV you can choose between several interpolation methods, see cv.Resize.

---

cv.GetRotationMatrix2D

Calculates the affine matrix of 2d rotation.

```
GetRotationMatrix2D(center,angle,scale,mapMatrix)-> None
```

center Center of the rotation in the source image

angle The rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner)

scale Isotropic scale factor

mapMatrix Pointer to the destination 2×3 matrix

The function `cv2DRotationMatrix` calculates the following matrix:

$$\begin{bmatrix} \alpha & \beta & (1-\alpha) \cdot \texttt{center.x} - \beta \cdot \texttt{center.y} \\ -\beta & \alpha & \beta \cdot \texttt{center.x} - (1-\alpha) \cdot \texttt{center.y} \end{bmatrix}$$

where

$$\alpha = \texttt{scale} \cdot cos(\texttt{angle}), \beta = \texttt{scale} \cdot sin(\texttt{angle})$$

The transformation maps the rotation center to itself. If this is not the purpose, the shift should be adjusted.

cv.GetAffineTransform

Calculates the affine transform from 3 corresponding points.

```
GetAffineTransform(src,dst,mapMatrix)-> None
```

src Coordinates of 3 triangle vertices in the source image

dst Coordinates of the 3 corresponding triangle vertices in the destination image

mapMatrix Pointer to the destination 2×3 matrix

The function cvGetAffineTransform calculates the matrix of an affine transform such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

cv.GetPerspectiveTransform

Calculates the perspective transform from 4 corresponding points.

```
GetPerspectiveTransform(src,dst,mapMatrix)-> None
```

src Coordinates of 4 quadrangle vertices in the source image

dst Coordinates of the 4 corresponding quadrangle vertices in the destination image

mapMatrix Pointer to the destination 3×3 matrix

The function `cvGetPerspectiveTransform` calculates a matrix of perspective transforms such that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

where

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

cv.GetQuadrangleSubPix

Retrieves the pixel quadrangle from an image with sub-pixel accuracy.

```
GetQuadrangleSubPix(src,dst,mapMatrix)-> None
```

src Source image

dst Extracted quadrangle

mapMatrix The transformation 2×3 matrix $[A|b]$ (see the discussion)

The function `cvGetQuadrangleSubPix` extracts pixels from `src` at sub-pixel accuracy and stores them to `dst` as follows:

$$dst(x, y) = src(A_{11}x' + A_{12}y' + b_1, A_{21}x' + A_{22}y' + b_2)$$

where

$$x' = x - \frac{(width(dst) - 1)}{2}, y' = y - \frac{(height(dst) - 1)}{2}$$

and

$$\texttt{mapMatrix} = \begin{bmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{bmatrix}$$

The values of pixels at non-integer coordinates are retrieved using bilinear interpolation. When the function needs pixels outside of the image, it uses replication border mode to reconstruct the values. Every channel of multiple-channel images is processed independently.

cv.GetRectSubPix

Retrieves the pixel rectangle from an image with sub-pixel accuracy.

```
GetRectSubPix(src,dst,center)-> None
```

src Source image

dst Extracted rectangle

center Floating point coordinates of the extracted rectangle center within the source image. The center must be inside the image

The function `cvGetRectSubPix` extracts pixels from `src`:

$$dst(x,y) = src(x + \texttt{center.x} - (width(\texttt{dst}) - 1) * 0.5, y + \texttt{center.y} - (height(\texttt{dst}) - 1) * 0.5)$$

where the values of the pixels at non-integer coordinates are retrieved using bilinear interpolation. Every channel of multiple-channel images is processed independently. While the rectangle center must be inside the image, parts of the rectangle may be outside. In this case, the replication border mode is used to get pixel values beyond the image boundaries.

---

cv.LogPolar

Remaps an image to log-polar space.

```
LogPolar(src,dst,center,M,flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS)->
None
```

src Source image

dst Destination image

center The transformation center; where the output precision is maximal

M Magnitude scale parameter. See below

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to zero

CV_WARP_INVERSE_MAP See below

The function `cvLogPolar` transforms the source image using the following transformation:
Forward transformation (`CV_WARP_INVERSE_MAP` is not set):

$$dst(\phi, \rho) = src(x, y)$$

Inverse transformation (`CV_WARP_INVERSE_MAP` is set):

$$dst(x, y) = src(\phi, \rho)$$

where

$$\rho = M \cdot \log \sqrt{x^2 + y^2}, \phi = atan(y/x)$$

The function emulates the human "foveal" vision and can be used for fast scale and rotation-invariant template matching, for object tracking and so forth. The function can not operate in-place.

---

cv.Remap

Applies a generic geometrical transformation to the image.

```
Remap(src,dst,mapx,mapy,flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS,fillval=(0,0,0,0))
None
```

src Source image

dst Destination image

mapx The map of x-coordinates (CV_32FC1 image)

mapy The map of y-coordinates (CV_32FC1 image)

flags A combination of interpolation method and the following optional flag(s):

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels. If some of them correspond to outliers in the source image, they are set to `fillval`

fillval A value used to fill outliers

The function `cvRemap` transforms the source image using the specified map:

$$\texttt{dst}(x, y) = \texttt{src}(\texttt{mapx}(x, y), \texttt{mapy}(x, y))$$

Similar to other geometrical transformations, some interpolation method (specified by user) is used to extract pixels with non-integer coordinates. Note that the function can not operate in-place.

cv.Resize

Resizes an image.

```
Resize(src,dst,interpolation=CV_INTER_LINEAR)-> None
```

src Source image

dst Destination image

interpolation Interpolation method:

- **CV_INTER_NN** nearest-neigbor interpolation
- **CV_INTER_LINEAR** bilinear interpolation (used by default)
- **CV_INTER_AREA** resampling using pixel area relation. It is the preferred method for image decimation that gives moire-free results. In terms of zooming it is similar to the CV_INTER_NN method
- **CV_INTER_CUBIC** bicubic interpolation

The function cvResize resizes an image src so that it fits exactly into dst. If ROI is set, the function considers the ROI as supported.

cv.WarpAffine

Applies an affine transformation to an image.

```
WarpAffine(src,dst,mapMatrix,flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,fillval=(0,0
None
```

src Source image

dst Destination image

mapMatrix 2×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`

CV_WARP_INVERSE_MAP indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpAffine` transforms the source image using the specified matrix:

$$dst(x', y') = src(x, y)$$

where

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV\_WARP\_INVERSE\_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

The function is similar to cv.GetQuadrangleSubPix but they are not exactly the same. cv.WarpAffine requires input and output image have the same data type, has larger overhead (so it is not quite suitable for small images) and can leave part of destination image unchanged. While cv.GetQuadrangleSubPix may extract quadrangles from 8-bit images into floating-point buffer, has smaller overhead and always changes the whole destination image content. Note that the function can not operate in-place.

To transform a sparse set of points, use the cv.Transform function from cxcore.

---

cv.WarpPerspective

Applies a perspective transformation to an image.

```
WarpPerspective(src,dst,mapMatrix,flags=CV_INNER_LINEAR+CV_WARP_FILL_OUTLIERS,fillval
None
```

src Source image

dst Destination image

mapMatrix 3×3 transformation matrix

flags A combination of interpolation methods and the following optional flags:

CV_WARP_FILL_OUTLIERS fills all of the destination image pixels; if some of them correspond to outliers in the source image, they are set to `fillval`

CV_WARP_INVERSE_MAP indicates that `matrix` is inversely transformed from the destination image to the source and, thus, can be used directly for pixel interpolation. Otherwise, the function finds the inverse transform from `mapMatrix`

fillval A value used to fill outliers

The function `cvWarpPerspective` transforms the source image using the specified matrix:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{if CV\_WARP\_INVERSE\_MAP is not set}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \texttt{mapMatrix} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \quad \text{otherwise}$$

Note that the function can not operate in-place. For a sparse set of points use the cv.PerspectiveTransform function from CxCore.

13.3 Miscellaneous Image Transformations

---

cv.AdaptiveThreshold

Applies an adaptive threshold to an array.

```
AdaptiveThreshold(src,dst,maxValue, adaptive_method=CV_ADAPTIVE_THRESH_MEAN_C,
thresholdType=CV_THRESH_BINARY,blockSize=3,param1=5)-> None
```

src Source image

dst Destination image

maxValue Maximum value that is used with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV`

adaptive_method Adaptive thresholding algorithm to use: `CV_ADAPTIVE_THRESH_MEAN_C` or `CV_ADAPTIVE_THRESH_GAUSSIAN_C` (see the discussion)

thresholdType Thresholding type; must be one of

CV_THRESH_BINARY xxx

CV_THRESH_BINARY_INV xxx

blockSize The size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on

param1 The method-dependent parameter. For the methods CV_ADAPTIVE_THRESH_MEAN_C and CV_ADAPTIVE_THRESH_GAUSSIAN_C it is a constant subtracted from the mean or weighted mean (see the discussion), though it may be negative

The function transforms a grayscale image to a binary image according to the formulas:

CV_THRESH_BINARY

$$dst(x,y) = \begin{cases} \texttt{maxValue} & \text{if } src(x,y) > T(x,y) \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0 & \text{if } src(x,y) > T(x,y) \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

where $T(x,y)$ is a threshold calculated individually for each pixel.

For the method CV_ADAPTIVE_THRESH_MEAN_C it is the mean of a blockSize × blockSize pixel neighborhood, minus param1.

For the method CV_ADAPTIVE_THRESH_GAUSSIAN_C it is the weighted sum (gaussian) of a blockSize × blockSize pixel neighborhood, minus param1.

---

cv.CvtColor

Converts an image from one color space to another.

```
CvtColor(src,dst,code)-> None
```

src The source 8-bit (8u), 16-bit (16u) or single-precision floating-point (32f) image

dst The destination image of the same data type as the source. The number of channels may be different

code Color conversion operation that can be specifed using `CV_ src_color_space 2 dst_color_space` constants (see below)

The function converts the input image from one color space to another. The function ignores the `colorModel` and `channelSeq` fields of the `IplImage` header, so the source image color space should be specified correctly (including order of the channels in the case of RGB space. For example, BGR means 24-bit format with $B_0, G_0, R_0, B_1, G_1, R_1, ...$ layout whereas RGB means 24-format with $R_0, G_0, B_0, R_1, G_1, B_1, ...$ layout).

The conventional range for R,G,B channel values is:

- 0 to 255 for 8-bit images
- 0 to 65535 for 16-bit images and
- 0 to 1 for floating-point images.

Of course, in the case of linear transformations the range can be specific, but in order to get correct results in the case of non-linear transformations, the input image should be scaled.

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as conversion to/from grayscale using:

  $$\text{RGB[A] to Gray:} Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

  and

  $$\text{Gray to RGB[A]:} R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

  The conversion from a RGB image to gray is done with:

  ```
  cvCvtColor(src ,bwsrc, CV_RGB2GRAY)
  ```

- RGB $\leftrightarrow$ CIE XYZ.Rec 709 with D65 white point (`CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV_XYZ2RGB`):

  $$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

  $$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

  X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB ↔ YCrCb JPEG (a.k.a. YCC) (`CV_BGR2YCrCb`, `CV_RGB2YCrCb`, `CV_YCrCb2BGR`, `CV_YCrCb2RGB`)

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$Cr \leftarrow (R - Y) \cdot 0.713 + delta$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + delta$$

$$R \leftarrow Y + 1.403 \cdot (Cr - delta)$$

$$G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta)$$

$$B \leftarrow Y + 1.773 \cdot (Cb - delta)$$

  where

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

  Y, Cr and Cb cover the whole value range.

- RGB ↔ HSV (`CV_BGR2HSV`, `CV_RGB2HSV`, `CV_HSV2BGR`, `CV_HSV2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B)/S & \text{if } V = R \\ 120 + 60(B - R)/S & \text{if } V = G \\ 240 + 60(R - G)/S & \text{if } V = B \end{cases}$$

  if $H < 0$ then $H \leftarrow H + 360$

  On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

  The values are then converted to the destination data type:

  8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

  16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

  32-bit images H, S, V are left as is

- RGB ↔ HLS (`CV_BGR2HLS, CV_RGB2HLS, CV_HLS2BGR, CV_HLS2RGB`). in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range.

$$V_{max} \leftarrow max(R,G,B)$$

$$V_{min} \leftarrow min(R,G,B)$$

$$L \leftarrow \frac{V_{max} - V_{min}}{2}$$

$$S \leftarrow \begin{cases} \frac{V_{max}-V_{min}}{V_{max}+V_{min}} & \text{if } L < 0.5 \\ \frac{V_{max}-V_{min}}{2-(V_{max}+V_{min})} & \text{if } L \geq 0.5 \end{cases}$$

$$H \leftarrow \begin{cases} 60(G-B)/S & \text{if } V_{max} = R \\ 120 + 60(B-R)/S & \text{if } V_{max} = G \\ 240 + 60(R-G)/S & \text{if } V_{max} = B \end{cases}$$

if $H < 0$ then $H \leftarrow H + 360$ On output $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$.

The values are then converted to the destination data type:

8-bit images

$$V \leftarrow 255V, S \leftarrow 255S, H \leftarrow H/2 \text{(to fit to 0 to 255)}$$

16-bit images (currently not supported)

$$V < -65535V, S < -65535S, H < -H$$

32-bit images H, S, V are left as is

- RGB ↔ CIE L*a*b* (`CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$X \leftarrow X/X_n, \text{where} X_n = 0.950456$$

$$Z \leftarrow Z/Z_n, \text{where} Z_n = 1.088754$$

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$

$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t <= 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output $0 \leq L \leq 100$, $-127 \leq a \leq 127$, $-127 \leq b \leq 127$

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow L * 255/100, a \leftarrow a + 128, b \leftarrow b + 128$$

16-bit images currently not supported

32-bit images L, a, b are left as is

- RGB $\leftrightarrow$ CIE L*u*v* (`CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB`) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$L \leftarrow \begin{cases} 116Y^{1/3} & \text{for } Y > 0.008856 \\ 903.3Y & \text{for } Y <= 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$

$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$

$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$

$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output $0 \leq L \leq 100$, $-134 \leq u \leq 220$, $-140 \leq v \leq 122$.

The values are then converted to the destination data type:

8-bit images

$$L \leftarrow 255/100L, u \leftarrow 255/354(u + 134), v \leftarrow 255/256(v + 140)$$

16-bit images currently not supported

32-bit images L, u, v are left as is

The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Ford98 at the Charles Poynton site.

- Bayer → RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerGB2RGB, CV_BayerRG2RGB, CV_BayerGR2RGB) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:

$$\begin{matrix} R & G & R & G & R \\ G & \mathrm{B} & \mathrm{G} & B & G \\ R & G & R & G & R \\ G & B & G & B & G \\ R & G & R & G & R \end{matrix}$$

The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C_1 and C_2 in the conversion constants CV_Bayer C_1C_2 2BGR and CV_Bayer C_1C_2 2RGB indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

---

cv.DistTransform

Calculates the distance to the closest zero pixel for all non-zero pixels of the source image.

```
DistTransform(src,dst,distance_type=CV_DIST_L2,mask_size=3,mask=None,labels=NULL)->
None
```

src 8-bit, single-channel (binary) source image

dst Output image with calculated distances (32-bit floating-point, single-channel)

distance_type Type of distance; can be CV_DIST_L1, CV_DIST_L2, CV_DIST_C or CV_DIST_USER

mask_size Size of the distance transform mask; can be 3 or 5. in the case of CV_DIST_L1 or CV_DIST_C the parameter is forced to 3, because a 3×3 mask gives the same result as a 5×5 yet it is faster

mask User-defined mask in the case of a user-defined distance, it consists of 2 numbers (horizontal/vertical shift cost, diagonal shift cost) in the case ofa 3×3 mask and 3 numbers (horizontal/vertical shift cost, diagonal shift cost, knight's move cost) in the case of a 5×5 mask

labels The optional output 2d array of integer type labels, the same size as `src` and `dst`

The function calculates the approximated distance from every binary image pixel to the nearest zero pixel. For zero pixels the function sets the zero distance, for others it finds the shortest path consisting of basic shifts: horizontal, vertical, diagonal or knight's move (the latest is available for a 5×5 mask). The overall distance is calculated as a sum of these basic distances. Because the distance function should be symmetric, all of the horizontal and vertical shifts must have the same cost (that is denoted as `a`), all the diagonal shifts must have the same cost (denoted `b`), and all knight's moves must have the same cost (denoted `c`). For `CV_DIST_C` and `CV_DIST_L1` types the distance is calculated precisely, whereas for `CV_DIST_L2` (Euclidian distance) the distance can be calculated only with some relative error (a 5×5 mask gives more accurate results), OpenCV uses the values suggested in [?]:

`CV_DIST_C`	(3×3)	a = 1, b = 1
`CV_DIST_L1`	(3×3)	a = 1, b = 2
`CV_DIST_L2`	(3×3)	a=0.955, b=1.3693
`CV_DIST_L2`	(5×5)	a=1, b=1.4, c=2.1969

And below are samples of the distance field (black (0) pixel is in the middle of white square) in the case of a user-defined distance:

User-defined 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1		1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

User-defined 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1		1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

Typically, for a fast, coarse distance estimation CV_DIST_L2, a 3×3 mask is used, and for a more accurate distance estimation CV_DIST_L2, a 5×5 mask is used.

When the output parameter labels is not NULL, for every non-zero pixel the function also finds the nearest connected component consisting of zero pixels. The connected components themselves are found as contours in the beginning of the function.

In this mode the processing time is still O(N), where N is the number of pixels. Thus, the function provides a very fast way to compute approximate Voronoi diagram for the binary image.

CvConnectedComp

Connected component, represented as a tuple (area, value, rect), where area is the area of the component as a float, value is the average color as a CvScalar , and rect is the ROI of the component, as a CvRect .

---

cv.FloodFill

Fills a connected component with the given color.

```
FloodFill(image,seed_point,new_val,lo_diff=(0,0,0,0),up_diff=(0,0,0,0),flags=4,mask=
comp
```

image Input 1- or 3-channel, 8-bit or floating-point image. It is modified by the function unless the CV_FLOODFILL_MASK_ONLY flag is set (see below)

seed_point The starting point

new_val New value of the repainted domain pixels

lo_diff Maximal lower brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

up_diff Maximal upper brightness/color difference between the currently observed pixel and one of its neighbors belonging to the component, or a seed pixel being added to the component. In the case of 8-bit color images it is a packed value

comp Returned connected component for the repainted domain

flags The operation flags. Lower bits contain connectivity value, 4 (by default) or 8, used within the function. Connectivity determines which neighbors of a pixel are considered. Upper bits can be 0 or a combination of the following flags:

CV_FLOODFILL_FIXED_RANGE if set, the difference between the current pixel and seed pixel is considered, otherwise the difference between neighbor pixels is considered (the range is floating)

CV_FLOODFILL_MASK_ONLY if set, the function does not fill the image (`new_val` is ignored), but fills the mask (that must be non-NULL in this case)

mask Operation mask, should be a single-channel 8-bit image, 2 pixels wider and 2 pixels taller than `image`. If not NULL, the function uses and updates the mask, so the user takes responsibility of initializing the `mask` content. Floodfilling can't go across non-zero pixels in the mask, for example, an edge detector output can be used as a mask to stop filling at edges. It is possible to use the same mask in multiple calls to the function to make sure the filled area do not overlap. **Note**: because the mask is larger than the filled image, a pixel in `mask` that corresponds to (x, y) pixel in `image` will have coordinates $(x + 1, y + 1)$

The function fills a connected component starting from the seed point with the specified color. The connectivity is determined by the closeness of pixel values. The pixel at (x, y) is considered to belong to the repainted domain if:

grayscale image, floating range

$$src(x', y') - \texttt{lo\_diff} <= src(x, y) <= src(x', y') + \texttt{up\_diff}$$

grayscale image, fixed range

$$src(seed.x, seed.y) - \texttt{lo\_diff} <= src(x, y) <= src(seed.x, seed.y) + \texttt{up\_diff}$$

color image, floating range

$$src(x', y')_r - \texttt{lo\_diff}_r <= src(x, y)_r <= src(x', y')_r + \texttt{up\_diff}_r$$
$$src(x', y')_g - \texttt{lo\_diff}_g <= src(x, y)_g <= src(x', y')_g + \texttt{up\_diff}_g$$
$$src(x', y')_b - \texttt{lo\_diff}_b <= src(x, y)_b <= src(x', y')_b + \texttt{up\_diff}_b$$

color image, fixed range

$$src(seed.x, seed.y)_r - \texttt{lo\_diff}_r <= src(x, y)_r <= src(seed.x, seed.y)_r + \texttt{up\_diff}_r$$
$$src(seed.x, seed.y)_g - \texttt{lo\_diff}_g <= src(x, y)_g <= src(seed.x, seed.y)_g + \texttt{up\_diff}_g$$
$$src(seed.x, seed.y)_b - \texttt{lo\_diff}_b <= src(x, y)_b <= src(seed.x, seed.y)_b + \texttt{up\_diff}_b$$

where $src(x', y')$ is the value of one of pixel neighbors. That is, to be added to the connected component, a pixel's color/brightness should be close enough to the:

- color/brightness of one of its neighbors that are already referred to the connected component in the case of floating range
- color/brightness of the seed point in the case of fixed range.

cv.Inpaint

Inpaints the selected region in the image.

```
Inpaint(src,mask,dst,inpaintRadius,flags) -> None
```

src The input 8-bit 1-channel or 3-channel image.

mask The inpainting mask, 8-bit 1-channel image. Non-zero pixels indicate the area that needs to be inpainted.

dst The output image of the same format and the same size as input.

inpaintRadius The radius of circlular neighborhood of each point inpainted that is considered by the algorithm.

flags The inpainting method, one of the following:

CV_INPAINT_NS Navier-Stokes based method.

CV_INPAINT_TELEA The method by Alexandru Telea [?]

The function reconstructs the selected image area from the pixel near the area boundary. The function may be used to remove dust and scratches from a scanned photo, or to remove undesirable objects from still images or video.

cv.Integral

Calculates the integral of an image.

```
Integral(image,sum,sqsum=NULL,tiltedSum=NULL)-> None
```

image The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

sum The integral image, $(W + 1) \times (H + 1)$, 32-bit integer or double precision floating-point (64f)

sqsum The integral image for squared pixel values, $(W + 1) \times (H + 1)$, double precision floating-point (64f)

tiltedSum The integral for the image rotated by 45 degrees, $(W+1) \times (H+1)$, the same data type as `sum`

The function calculates one or more integral images for the source image as following:

$$\texttt{sum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)$$

$$\texttt{sqsum}(X,Y) = \sum_{x<X,y<Y} \texttt{image}(x,y)^2$$

$$\texttt{tiltedSum}(X,Y) = \sum_{y<Y,abs(x-X)<y} \texttt{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

$$\sum_{x_1<=x<x_2,\, y_1<=y<y_2} = \texttt{sum}(x_2,y_2) - \texttt{sum}(x_1,y_2) - \texttt{sum}(x_2,y_1) + \texttt{sum}(x_1,x_1)$$

It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

---

cv.PyrMeanShiftFiltering

Does meanshift image segmentation

```
PyrMeanShiftFiltering(src,dst,sp,sr,max_level=1,termcrit=(CV_TERMCRIT_ITER+CV_TERMCR
None
```

src The source 8-bit, 3-channel image.

dst The destination image of the same format and the same size as the source.

sp The spatial window radius.

sr The color window radius.

max_level Maximum level of the pyramid for the segmentation.

termcrit Termination criteria: when to stop meanshift iterations.

The function implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel (X,Y) neighborhood in the joint space-color hyperspace is considered:

$$(x,y): X-\texttt{sp} \leq x \leq X+\texttt{sp}, Y-\texttt{sp} \leq y \leq Y+\texttt{sp}, ||(R,G,B)-(r,g,b)|| \leq \texttt{sr}$$

where `(R,G,B)` and `(r,g,b)` are the vectors of color components at `(X,Y)` and `(x,y)`, respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value `(X',Y')` and average color vector `(R',G',B')` are found and they act as the neighborhood center on the next iteration:

(X,Y) (X',Y'), (R,G,B) (R',G',B').

After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration):

$I(X,Y) <- (R*,G*,B*)$

Then `max_level` > 0 , the gaussian pyramid of `max_level` $+ 1$ levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much ($>$ `sr`) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when `max_level` $== 0$).

---

cv.PyrSegmentation

Implements image segmentation by pyramids.

```
PyrSegmentation(src,dst,storage,level,threshold1,threshold2)-> comp
```

src The source image

dst The destination image

storage Storage; stores the resulting sequence of connected components

comp Pointer to the output sequence of the segmented components

level Maximum level of the pyramid for the segmentation

threshold1 Error threshold for establishing the links

threshold2 Error threshold for the segments clustering

The function implements image segmentation by pyramids. The pyramid builds up to the level `level`. The links between any pixel `a` on level `i` and its candidate father pixel `b` on the adjacent level are established if $p(c(a), c(b)) < threshold1$. After the connected components are defined, they are joined into several clusters. Any two segments A and B belong to the same cluster, if $p(c(A), c(B)) < threshold2$. If the input image has only one channel, then $p(c^1, c^2) = |c^1 - c^2|$. If the input image has three channels (red, green and blue), then

$$p(c^1, c^2) = 0.30(c^1_r - c^2_r) + 0.59(c^1_g - c^2_g) + 0.11(c^1_b - c^2_b).$$

There may be more than one connected component per a cluster. The images `src` and `dst` should be 8-bit single-channel or 3-channel images or equal size.

---

cv.Threshold

Applies a fixed-level threshold to array elements.

```
Threshold(src,dst,threshold,maxValue,thresholdType)-> None
```

src Source array (single-channel, 8-bit or 32-bit floating point)

dst Destination array; must be either the same type as `src` or 8-bit

threshold Threshold value

maxValue Maximum value to use with `CV_THRESH_BINARY` and `CV_THRESH_BINARY_INV` thresholding types

thresholdType Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image (cv.CmpS could be also used for this purpose) or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by `thresholdType`:

CV_THRESH_BINARY

$$\texttt{dst}(x,y) = \begin{cases} \texttt{maxValue} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_BINARY_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{maxValue} & \text{otherwise} \end{cases}$$

CV_THRESH_TRUNC

$$\texttt{dst}(x,y) = \begin{cases} \texttt{threshold} & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

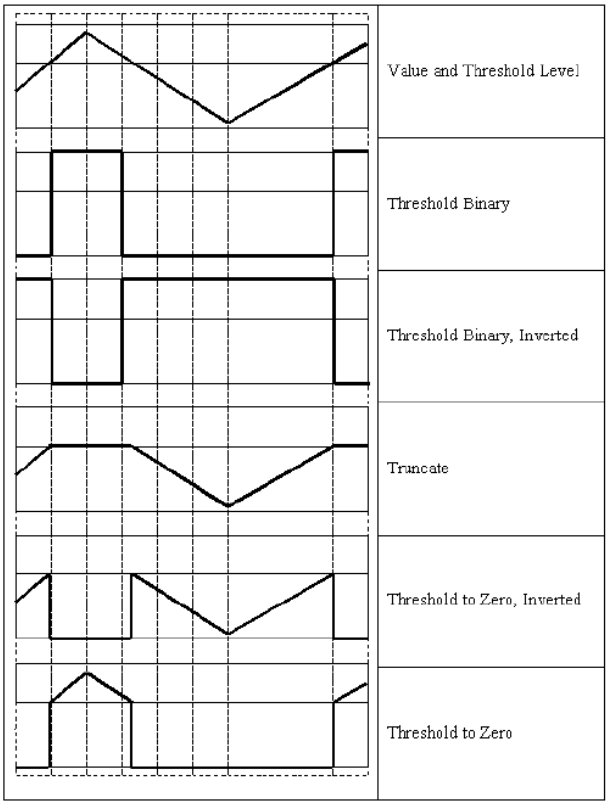
CV_THRESH_TOZERO

$$\texttt{dst}(x,y) = \begin{cases} \texttt{src}(x,y) & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ 0 & \text{otherwise} \end{cases}$$

CV_THRESH_TOZERO_INV

$$\texttt{dst}(x,y) = \begin{cases} 0 & \text{if } \texttt{src}(x,y) > \texttt{threshold} \\ \texttt{src}(x,y) & \text{otherwise} \end{cases}$$

Also, the special value CV_THRESH_OTSU may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified thresh. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.



13.4 Histograms

CvHistogram

Multi-dimensional histogram.

A CvHistogram is a multi-dimensional histogram, created by function CreateHist . It has an attribute `bins` a CvMatND containing the histogram counts.

---

cv.CalcBackProject

Calculates the back projection.

```
CalcBackProject(image,back_project,hist)-> None
```

image Source images (though you may pass CvMat** as well)

back_project Destination back projection image of the same type as the source images

hist Histogram

The function calculates the back project of the histogram. For each tuple of pixels at the same position of all input single-channel images the function puts the value of the histogram bin, corresponding to the tuple in the destination image. In terms of statistics, the value of each output image pixel is the probability of the observed tuple given the distribution (histogram). For example, to find a red object in the picture, one may do the following:

1. Calculate a hue histogram for the red object assuming the image contains only this object. The histogram is likely to have a strong maximum, corresponding to red color.
2. Calculate back projection of a hue plane of input image where the object is searched, using the histogram. Threshold the image.
3. Find connected components in the resulting picture and choose the right component using some additional criteria, for example, the largest connected component.

That is the approximate algorithm of Camshift color object tracker, except for the 3rd step, instead of which CAMSHIFT algorithm is used to locate the object on the back projection given the previous object position.

---

cv.CalcBackProjectPatch

Locates a template within an image by using a histogram comparison.

```
CalcBackProjectPatch(images,dst,patch_size,hist,method,factor)-> None
```

images Source images (though, you may pass CvMat** as well)

dst Destination image

patch_size Size of the patch slid though the source image

hist Histogram

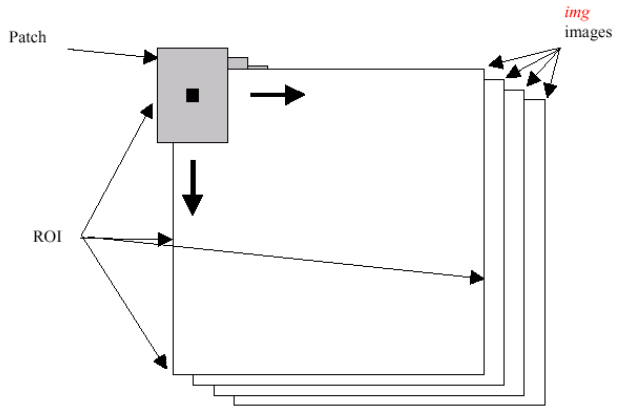
method Compasion method, passed to cv.CompareHist (see description of that function)

factor Normalization factor for histograms, will affect the normalization scale of the destination image, pass 1 if unsure

The function calculates the back projection by comparing histograms of the source image patches with the given histogram. Taking measurement results from some image at each location over ROI creates an array `image`. These results might be one or more of hue, `x` derivative, `y` derivative, Laplacian filter, oriented Gabor filter, etc. Each measurement output is collected into its own separate image. The `image` image array is a collection of these measurement images. A multi-dimensional histogram `hist` is constructed by sampling from the `image` image array. The final histogram is normalized. The `hist` histogram has as many dimensions as the number of elements in `image` array.

Each new image is measured and then converted into an `image` image array over a chosen ROI. Histograms are taken from this `image` image in an area covered by a "patch" with an anchor at center as shown in the picture below. The histogram is normalized using the parameter `norm_factor` so that it may be compared with `hist`. The calculated histogram is compared to the model histogram; `hist` uses The function `cvCompareHist` with the comparison method=`method`). The resulting output is placed at the location corresponding to the patch anchor in the probability image `dst`. This process is repeated as the patch is slid over the ROI. Iterative histogram update by subtracting trailing pixels covered by the patch and adding newly covered pixels to the histogram can save a lot of operations, though it is not implemented yet.

Back Project Calculation by Patches



---

cv.CalcHist

Calculates the histogram of image(s).

```
CalcHist(image,hist,accumulate=0,mask=NULL)-> None
```

image Source images (though you may pass CvMat** as well)

hist Pointer to the histogram

accumulate Accumulation flag. If it is set, the histogram is not cleared in the beginning. This feature allows user to compute a single histogram from several images, or to update the histogram online

mask The operation mask, determines what pixels of the source images are counted

The function calculates the histogram of one or more single-channel images. The elements of a tuple that is used to increment a histogram bin are taken at the same location from the corresponding input images.

---

cv.CalcProbDensity

Divides one histogram by another.

```
CalcProbDensity(hist1,hist2,dst_hist,scale=255)-> None
```

hist1 first histogram (the divisor)

hist2 second histogram

dst_hist destination histogram

scale scale factor for the destination histogram

The function calculates the object probability density from the two histograms as:

$$\texttt{dist\_hist}(I) = \begin{cases} 0 & \text{if } \texttt{hist1}(I) = 0 \\ \texttt{scale} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) > \texttt{hist1}(I) \\ \frac{\texttt{hist2}(I) \cdot \texttt{scale}}{\texttt{hist1}(I)} & \text{if } \texttt{hist1}(I) \neq 0 \text{ and } \texttt{hist2}(I) \leq \texttt{hist1}(I) \end{cases}$$

So the destination histogram bins are within less than `scale`.

---

cv.ClearHist

Clears the histogram.

```
ClearHist(hist)-> None
```

hist Histogram

The function sets all of the histogram bins to 0 in the case of a dense histogram and removes all histogram bins in the case of a sparse array.

---

cv.CompareHist

Compares two dense histograms.

```
CompareHist(hist1,hist2,method)->float
```

hist1 The first dense histogram

hist2 The second dense histogram

method Comparison method, one of the following:

CV_COMP_CORREL Correlation

CV_COMP_CHISQR Chi-Square

CV_COMP_INTERSECT Intersection

CV_COMP_BHATTACHARYYA Bhattacharyya distance

The function compares two dense histograms using the specified method (H_1 denotes the first histogram, H_2 the second):

Correlation (method=CV_COMP_CORREL)

$$d(H_1,H_2)=\frac{\sum_I (H'_1(I)\cdot H'_2(I))}{\sqrt{\sum_I (H'_1(I)^2)\cdot \sum_I (H'_2(I)^2)}}$$

where

$$H'_k(I)=\frac{H_k(I)-1}{N\cdot \sum_J H_k(J)}$$

where N is the number of histogram bins.

Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1, H_2) = \sum_I \frac{(H_1(I) - H_2(I))^2}{H_1(I) + H_2(I)}$$

Intersection (method=CV_COMP_INTERSECT)

$$d(H_1, H_2) = \sum_I \min(H_1(I), H_2(I))$$

Bhattacharyya distance (method=CV_COMP_BHATTACHARYYA)

$$d(H_1, H_2) = \sqrt{1 - \sum_I \frac{\sqrt{H_1(I) \cdot H_2(I)}}{\sqrt{\sum_I H_1(I) \cdot \sum_I H_2(I)}}}$$

The function returns $d(H_1, H_2)$.

Note: the method `CV_COMP_BHATTACHARYYA` only works with normalized histograms.

To compare a sparse histogram or more general sparse configurations of weighted points, consider using the cv.CalcEMD2 function.

---

cv.CreateHist

Creates a histogram.

```
CreateHist(dims, type, ranges, uniform = 1) -> hist
```

dims for an N-dimensional histogram, list of length N giving the size of each dimension

type Histogram representation format: `CV_HIST_ARRAY` means that the histogram data is represented as a multi-dimensional dense array CvMatND; `CV_HIST_SPARSE` means that histogram data is represented as a multi-dimensional sparse array CvSparseMat

ranges Array of ranges for the histogram bins. Its meaning depends on the `uniform` parameter value. The ranges are used for when the histogram is calculated or backprojected to determine which histogram bin corresponds to which value/tuple of values from the input image(s)

uniform Uniformity flag; if not 0, the histogram has evenly spaced bins and for every $0 <= i < cDims$ `ranges[i]` is an array of two numbers: lower and upper boundaries for the i-th histogram dimension. The whole range [lower,upper] is then split into `dims[i]` equal parts to determine the `i-th` input tuple value ranges for every histogram bin. And if `uniform=0`, then `i-th` element of `ranges` array contains `dims[i]+1` elements: $\texttt{lower}_0, \texttt{upper}_0, \texttt{lower}_1, \texttt{upper}_1 = \texttt{lower}_2, ...\texttt{upper}_{dims[i]-1}$ where $\texttt{lower}_j$ and $\texttt{upper}_j$ are lower and upper boundaries of `i-th` input tuple value for `j-th` bin, respectively. In either case, the input values that are beyond the specified range for a histogram bin are not counted by cv.CalcHist and filled with 0 by cv.CalcBackProject

The function creates a histogram of the specified size and returns a pointer to the created histogram. If the array `ranges` is 0, the histogram bin ranges must be specified later via the function cv.SetHistBinRanges. Though cv.CalcHist and cv.CalcBackProject may process 8-bit images without setting bin ranges, they assume thy are equally spaced in 0 to 255 bins.

---

cv.GetMinMaxHistValue

Finds the minimum and maximum histogram bins.

```
GetMinMaxHistValue(hist)-> (min_value,max_value,min_idx,max_idx)
```

hist Histogram

min_value Minimum value of the histogram

max_value Maximum value of the histogram

min_idx Coordinates of the minimum

max_idx Coordinates of the maximum

The function finds the minimum and maximum histogram bins and their positions. All of output arguments are optional. Among several extremas with the same value the ones with the minimum index (in lexicographical order) are returned. In the case of several maximums or minimums, the earliest in lexicographical order (extrema locations) is returned.

---

cv.NormalizeHist

Normalizes the histogram.

```
NormalizeHist(hist,factor)-> None
```

hist Pointer to the histogram

factor Normalization factor

The function normalizes the histogram bins by scaling them, such that the sum of the bins becomes equal to `factor`.

QueryHistValue_1D

Returns the value from a 1D histogram bin.

```
QueryHistValue_1D(hist, idx0) -> float
```

hist Histogram

idx0 bin index 0

QueryHistValue_2D

Returns the value from a 2D histogram bin.

```
QueryHistValue_2D(hist, idx0, idx1) -> float
```

hist Histogram

idx0 bin index 0

idx1 bin index 1

QueryHistValue_3D

Returns the value from a 3D histogram bin.

```
QueryHistValue_3D(hist, idx0, idx1, idx2) -> float
```

hist Histogram

idx0 bin index 0

idx1 bin index 1

idx2 bin index 2

QueryHistValue_nD

Returns the value from a 1D histogram bin.

```
QueryHistValue_nD(hist, idx) -> float
```

hist Histogram

idx list of indices, of same length as the dimension of the histogram's bin.

---

cv.ThreshHist

Thresholds the histogram.

```
ThreshHist(hist,threshold)-> None
```

hist Pointer to the histogram

threshold Threshold level

The function clears histogram bins that are below the specified threshold.

13.5 Feature Detection

cv.Canny

Implements the Canny algorithm for edge detection.

```
Canny(image,edges,threshold1,threshold2,aperture_size=3)-> None
```

image Single-channel input image

edges Single-channel image to store the edges found by the function

threshold1 The first threshold

threshold2 The second threshold

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel)

The function finds the edges on the input image `image` and marks them in the output image `edges` using the Canny algorithm. The smallest value between `threshold1` and `threshold2` is used for edge linking, the largest value is used to find the initial segments of strong edges.

cv.CornerEigenValsAndVecs

Calculates eigenvalues and eigenvectors of image blocks for corner detection.

```
CornerEigenValsAndVecs(image,eigenvv,blockSize,aperture_size=3)-> None
```

image Input image

eigenvv Image to store the results. It must be 6 times wider than the input image

blockSize Neighborhood size (see discussion)

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel)

For every pixel, the function `cvCornerEigenValsAndVecs` considers a `blockSize×blockSize` neigborhood S(p). It calcualtes the covariation matrix of derivatives over the neigborhood as:

$$M = \begin{bmatrix} \sum_{S(p)}(dI/dx)^2 & \sum_{S(p)}(dI/dx \cdot dI/dy)^2 \\ \sum_{S(p)}(dI/dx \cdot dI/dy)^2 & \sum_{S(p)}(dI/dy)^2 \end{bmatrix}$$

After that it finds eigenvectors and eigenvalues of the matrix and stores them into destination image in form $(\lambda_1, \lambda_2, x_1, y_1, x_2, y_2)$ where

λ_1, λ_2 are the eigenvalues of M; not sorted

x_1, y_1 are the eigenvectors corresponding to λ_1

x_2, y_2 are the eigenvectors corresponding to λ_2

---

cv.CornerHarris

Harris edge detector.

```
CornerHarris(image,harris_dst,blockSize,aperture_size=3,k=0.04)-> None
```

image Input image

harris_dst Image to store the Harris detector responses. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cv.CornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel).

k Harris detector free parameter. See the formula below

The function runs the Harris edge detector on the image. Similarly to cv.CornerMinEigenVal and cv.CornerEigenValsAndVecs, for each pixel it calculates a 2×2 gradient covariation matrix M over a `blockSize` × `blockSize` neighborhood. Then, it stores

$$det(M) - k\, trace(M)^2$$

to the destination image. Corners in the image can be found as the local maxima of the destination image.

cv.CornerMinEigenVal

Calculates the minimal eigenvalue of gradient matrices for corner detection.

```
CornerMinEigenVal(image,eigenval,blockSize,aperture_size=3)-> None
```

image Input image

eigenval Image to store the minimal eigenvalues. Should have the same size as `image`

blockSize Neighborhood size (see the discussion of cv.CornerEigenValsAndVecs)

aperture_size Aperture parameter for the Sobel operator (see cv.Sobel).

The function is similar to cv.CornerEigenValsAndVecs but it calculates and stores only the minimal eigen value of derivative covariation matrix for every pixel, i.e. $min(\lambda_1, \lambda_2)$ in terms of the previous function.

cv.ExtractSURF

Extracts Speeded Up Robust Features from an image.

```
ExtractSURF(image,mask,storage,params)-> (keypoints,descriptors)
```

image The input 8-bit grayscale image

mask The optional input 8-bit mask. The features are only found in the areas that contain more than 50% of non-zero mask pixels

keypoints The output parameter; double pointer to the sequence of keypoints. The sequence of CvSURFPoint structures is as follows:

```
typedef struct CvSURFPoint
{
   CvPoint2D32f pt; // position of the feature within the image
   int laplacian;   // -1, 0 or +1. sign of the laplacian at the point.
                    // can be used to speedup feature comparison
                    // (normally features with laplacians of different
            // signs can not match)
```

```
    int size;        // size of the feature
    float dir;       // orientation of the feature: 0..360 degrees
    float hessian;   // value of the hessian (can be used to
             // approximately estimate the feature strengths;
                     // see also params.hessianThreshold)
}
CvSURFPoint;
```

descriptors The optional output parameter; double pointer to the sequence of descriptors. Depending on the params.extended value, each element of the sequence will be either a 64-element or a 128-element floating-point (CV_32F) vector. If the parameter is NULL, the descriptors are not computed

storage Memory storage where keypoints and descriptors will be stored

params Various algorithm parameters put to the structure CvSURFParams:

```
typedef struct CvSURFParams
{
    int extended; // 0 means basic descriptors (64 elements each),
                  // 1 means extended descriptors (128 elements each)
    double hessianThreshold; // only features with keypoint.hessian
          // larger than that are extracted.
                  // good default value is ~300-500 (can depend on the
          // average local contrast and sharpness of the image).
                  // user can further filter out some features based on
          // their hessian values and other characteristics.
    int nOctaves; // the number of octaves to be used for extraction.
                  // With each next octave the feature size is doubled
          // (3 by default)
    int nOctaveLayers; // The number of layers within each octave
          // (4 by default)
}
CvSURFParams;

CvSURFParams cvSURFParams(double hessianThreshold, int extended=0);
          // returns default parameters
```

The function cvExtractSURF finds robust features in the image, as described in Bay06 . For each feature it returns its location, size, orientation and optionally the descriptor, basic or extended. The function can be used for object tracking and localization, image stitching etc. See the find_obj.cpp demo in OpenCV samples directory.

cv.FindCornerSubPix

Refines the corner locations.

```
FindCornerSubPix(image,corners,win,zero_zone,criteria)-> corners
```

image Input image

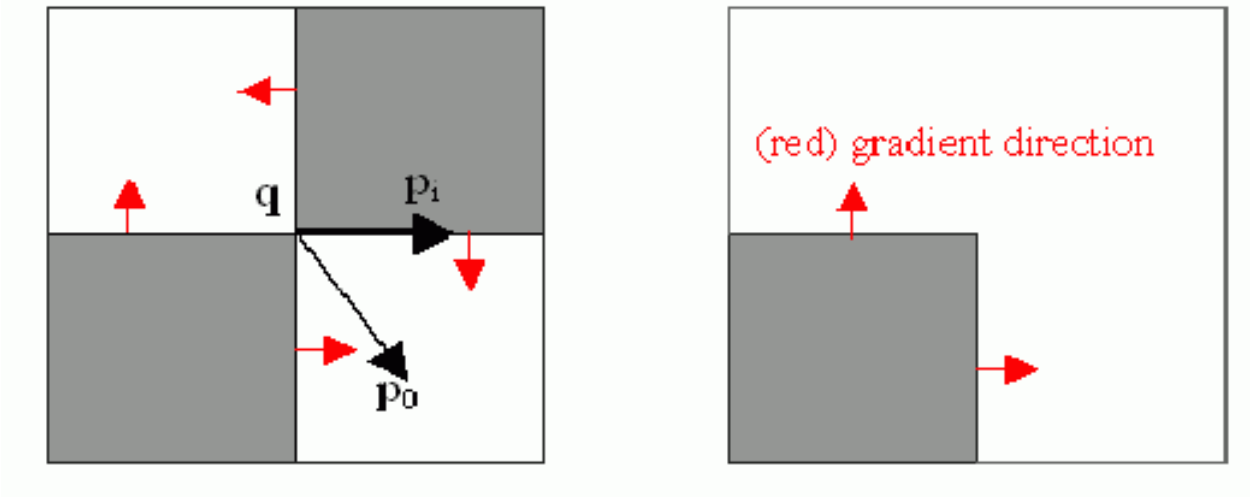
corners Initial coordinates of the input corners as a list of (x, y) pairs

win Half of the side length of the search window. For example, if `win`=(5,5), then a $5*2+1 \times 5*2+1 = 11 \times 11$ search window would be used

zero_zone Half of the size of the dead region in the middle of the search zone over which the summation in the formula below is not done. It is used sometimes to avoid possible singularities of the autocorrelation matrix. The value of (-1,-1) indicates that there is no such size

criteria Criteria for termination of the iterative process of corner refinement. That is, the process of corner position refinement stops either after a certain number of iterations or when a required accuracy is achieved. The `criteria` may specify either of or both the maximum number of iteration and the required accuracy

The function iterates to find the sub-pixel accurate location of corners, or radial saddle points, as shown in on the picture below. It returns the refined coordinates as a list of (x, y) pairs.



Sub-pixel accurate corner locator is based on the observation that every vector from the center q to a point p located within a neighborhood of q is orthogonal to the image gradient at p subject to image and measurement noise. Consider the expression:

$$\epsilon_i = {DI_{p_i}}^T \cdot (q - p_i)$$

where DI_{p_i} is the image gradient at the one of the points p_i in a neighborhood of q. The value of q is to be found such that ϵ_i is minimized. A system of equations may be set up with ϵ_i set to zero:

$$\sum_i (DI_{p_i} \cdot {DI_{p_i}}^T)q = \sum_i (DI_{p_i} \cdot {DI_{p_i}}^T \cdot p_i)$$

where the gradients are summed within a neighborhood ("search window") of q. Calling the first gradient term G and the second gradient term b gives:

$$q = G^{-1} \cdot b$$

The algorithm sets the center of the neighborhood window at this new center q and then iterates until the center keeps within a set threshold.

---

cv.GetStarKeypoints

Retrieves keypoints using the StarDetector algorithm.

```
GetStarKeypoints(image,storage,params)-> keypoints
```

image The input 8-bit grayscale image

storage Memory storage where the keypoints will be stored

params Various algorithm parameters given to the structure CvStarDetectorParams:

```
typedef struct CvStarDetectorParams
{
   int maxSize; // maximal size of the features detected. The following
                // values of the parameter are supported:
                // 4, 6, 8, 11, 12, 16, 22, 23, 32, 45, 46, 64, 90, 128
   int responseThreshold; // threshold for the approximatd laplacian,
                          // used to eliminate weak features
   int lineThresholdProjected; // another threshold for laplacian to
               // eliminate edges
```

```
    int lineThresholdBinarized; // another threshold for the feature
                // scale to eliminate edges
    int suppressNonmaxSize; // linear size of a pixel neighborhood
                // for non-maxima suppression
}
CvStarDetectorParams;
```

The function GetStarKeypoints extracts keypoints that are local scale-space extremas. The scale-space is constructed by computing approximate values of laplacians with different sigma's at each pixel. Instead of using pyramids, a popular approach to save computing time, all of the laplacians are computed at each pixel of the original high-resolution image. But each approximate laplacian value is computed in O(1) time regardless of the sigma, thanks to the use of integral images. The algorithm is based on the paper Agrawal08 , but instead of a square, hexagon or octagon it uses an 8-end star shape, hence the name, consisting of overlapping upright and tilted squares.

Each computed feature is represented by the following structure:

```
typedef struct CvStarKeypoint
{
    CvPoint pt; // coordinates of the feature
    int size; // feature size, see CvStarDetectorParams::maxSize
    float response; // the approximated laplacian value at that point.
}
CvStarKeypoint;

inline CvStarKeypoint cvStarKeypoint(CvPoint pt, int size, float response);
```

---

cv.GoodFeaturesToTrack

Determines strong corners on an image.

```
GoodFeaturesToTrack(image,eigImage,tempImage,cornerCount,qualityLevel,minDistance,m
corners
```

image The source 8-bit or floating-point 32-bit, single-channel image

eigImage Temporary floating-point 32-bit image, the same size as `image`

tempImage Another temporary image, the same size and format as `eigImage`

cornerCount number of corners to detect

qualityLevel Multiplier for the max/min eigenvalue; specifies the minimal accepted quality of image corners

minDistance Limit, specifying the minimum possible distance between the returned corners; Euclidian distance is used

mask Region of interest. The function selects points either in the specified region or in the whole image if the mask is NULL

blockSize Size of the averaging block, passed to the underlying cv.CornerMinEigenVal or cv.CornerHarris used by the function

useHarris If nonzero, Harris operator (cv.CornerHarris) is used instead of default cv.CornerMinEigenVal

k Free parameter of Harris detector; used only if ($\texttt{useHarris} != 0$)

The function finds the corners with big eigenvalues in the image. The function first calculates the minimal eigenvalue for every source image pixel using the cv.CornerMinEigenVal function and stores them in `eigImage`. Then it performs non-maxima suppression (only the local maxima in 3×3 neighborhood are retained). The next step rejects the corners with the minimal eigenvalue less than $\texttt{qualityLevel} \cdot max(\texttt{eigImage}(x, y))$. Finally, the function ensures that the distance between any two corners is not smaller than `minDistance`. The weaker corners (with a smaller min eigenvalue) that are too close to the stronger corners are rejected.

Note that the if the function is called with different values `A` and `B` of the parameter `qualityLevel`, and `A` ¿ B, the array of returned corners with `qualityLevel=A` will be the prefix of the output corners array with `qualityLevel=B`.

---

cv.HoughLines2

Finds lines in a binary image using a Hough transform.

```
HoughLines2(image,storage,method,rho,theta,threshold,param1=0,param2=0)->
lines
```

image The 8-bit, single-channel, binary source image. In the case of a probabilistic method, the image is modified by the function

storage The storage for the lines that are detected. It can be a memory storage (in this case a sequence of lines is created in the storage and returned by the function) or single row/single column matrix (CvMat*) of a particular type (see below) to which the lines' parameters are

written. The matrix header is modified by the function so its `cols` or `rows` will contain the number of lines detected. If `storage` is a matrix and the actual number of lines exceeds the matrix size, the maximum possible number of lines is returned (in the case of standard hough transform the lines are sorted by the accumulator value)

method The Hough transform variant, one of the following:

CV_HOUGH_STANDARD classical or standard Hough transform. Every line is represented by two floating-point numbers (ρ, θ), where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of `CV_32FC2` type

CV_HOUGH_PROBABILISTIC probabilistic Hough transform (more efficient in case if picture contains a few long linear segments). It returns line segments rather than the whole line. Each segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of `CV_32SC4` type

CV_HOUGH_MULTI_SCALE multi-scale variant of the classical Hough transform. The lines are encoded the same way as `CV_HOUGH_STANDARD`

rho Distance resolution in pixel-related units

theta Angle resolution measured in radians

threshold Threshold parameter. A line is returned by the function if the corresponding accumulator value is greater than `threshold`

param1 The first method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the minimum line length.
- For the multi-scale Hough transform it is the divisor for the distance resolution ρ. (The coarse distance resolution will be ρ and the accurate resolution will be $(\rho/\texttt{param1})$).

param2 The second method-dependent parameter:

- For the classical Hough transform it is not used (0).
- For the probabilistic Hough transform it is the maximum gap between line segments lying on the same line to treat them as a single line segment (i.e. to join them).
- For the multi-scale Hough transform it is the divisor for the angle resolution θ. (The coarse angle resolution will be θ and the accurate resolution will be $(\theta/\texttt{param2})$).

The function implements a few variants of the Hough transform for line detection.

cv.PreCornerDetect

Calculates the feature map for corner detection.

```
PreCornerDetect(image,corners,apertureSize=3)-> None
```

image Input image

corners Image to store the corner candidates

apertureSize Aperture parameter for the Sobel operator (see cv.Sobel)

The function calculates the function

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}$$

where $D_?$ denotes one of the first image derivatives and $D_{??}$ denotes a second image derivative.

The corners can be found as local maximums of the function below:

```
// assume that the image is floating-point
IplImage* corners = cvCloneImage(image);
IplImage* dilated_corners = cvCloneImage(image);
IplImage* corner_mask = cvCreateImage( cvGetSize(image), 8, 1 );
cvPreCornerDetect( image, corners, 3 );
cvDilate( corners, dilated_corners, 0, 1 );
cvSubS( corners, dilated_corners, corners );
cvCmpS( corners, 0, corner_mask, CV_CMP_GE );
cvReleaseImage( &corners );
cvReleaseImage( &dilated_corners );
```

13.6 Motion Analysis and Object Tracking

cv.Acc

Adds a frame to an accumulator.

```
Acc(image,sum,mask=NULL)-> None
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point. (each channel of multi-channel image is processed independently)

sum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the whole image `image` or its selected region to the accumulator `sum`:

$$\texttt{sum}(x,y) \leftarrow \texttt{sum}(x,y) + \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cv.CalcGlobalOrientation

Calculates the global motion orientation of some selected region.

```
CalcGlobalOrientation(orientation,mask,mhi,timestamp,duration)-> float
```

orientation Motion gradient orientation image; calculated by the function cv.CalcMotionGradient

mask Mask image. It may be a conjunction of a valid gradient mask, obtained with cv.CalcMotionGradient and the mask of the region, whose direction needs to be calculated

mhi Motion history image

timestamp Current time in milliseconds or other units, it is better to store time passed to cv.UpdateMotionHistory before and reuse it here, because running cv.UpdateMotionHistory and cv.CalcMotionGradient on large images may take some time

duration Maximal duration of motion track in milliseconds, the same as cv.UpdateMotionHistory

The function calculates the general motion direction in the selected region and returns the angle between 0 degrees and 360 degrees . At first the function builds the orientation histogram and finds the basic orientation as a coordinate of the histogram maximum. After that the function calculates the shift relative to the basic orientation as a weighted sum of all of the orientation vectors: the more recent the motion, the greater the weight. The resultant angle is a circular sum of the basic orientation and the shift.

---

cv.CalcMotionGradient

Calculates the gradient orientation of a motion history image.

```
CalcMotionGradient(mhi,mask,orientation,delta1,delta2,apertureSize=3)->
None
```

mhi Motion history image

mask Mask image; marks pixels where the motion gradient data is correct; output parameter

orientation Motion gradient orientation image; contains angles from 0 to 360 degrees

delta1 See below

delta2 See below

apertureSize Aperture size of derivative operators used by the function: CV_SCHARR, 1, 3, 5 or 7 (see cv.Sobel)

The function calculates the derivatives Dx and Dy of `mhi` and then calculates gradient orientation as:

$$\texttt{orientation}(x,y) = \arctan \frac{Dy(x,y)}{Dx(x,y)}$$

where both $Dx(x,y)$ and $Dy(x,y)$ signs are taken into account (as in the cv.CartToPolar function). After that `mask` is filled to indicate where the orientation is valid (see the `delta1` and `delta2` description).

The function finds the minimum ($m(x,y)$) and maximum ($M(x,y)$) mhi values over each pixel (x,y) neighborhood and assumes the gradient is valid only if

$$\min(\texttt{delta1},\texttt{delta2}) \leq M(x,y) - m(x,y) \leq \max(\texttt{delta1},\texttt{delta2}).$$

---

cv.CalcOpticalFlowBM

Calculates the optical flow for two images by using the block matching method.

```
CalcOpticalFlowBM(prev,curr,blockSize,shiftSize,max_range,usePrevious,velx,vely)->
None
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

blockSize Size of basic blocks that are compared

shiftSize Block coordinate increments

max_range Size of the scanned neighborhood in pixels around the block

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of

$$\left\lfloor \frac{\texttt{prev->width} - \texttt{blockSize.width}}{\texttt{shiftSize.width}} \right\rfloor \times \left\lfloor \frac{\texttt{prev->height} - \texttt{blockSize.height}}{\texttt{shiftSize.height}} \right\rfloor$$

size, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size `velx`, 32-bit floating-point, single-channel

The function calculates the optical flow for overlapped blocks `blockSize.width×blockSize.height` pixels each, thus the velocity fields are smaller than the original images. For every block in `prev` the functions tries to find a similar block in `curr` in some neighborhood of the original block or shifted by (velx(x0,y0),vely(x0,y0)) block as has been calculated by previous function call (if `usePrevious=1`)

---

cv.CalcOpticalFlowHS

Calculates the optical flow for two images.

```
CalcOpticalFlowHS(prev,curr,usePrevious,velx,vely,lambda,criteria)->
None
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

usePrevious Uses the previous (input) velocity field

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

lambda Lagrangian multiplier

criteria Criteria of termination of velocity computing

The function computes the flow for every pixel of the first input image using the Horn and Schunck algorithm [?].

---

cv.CalcOpticalFlowLK

Calculates the optical flow for two images.

```
CalcOpticalFlowLK(prev,curr,winSize,velx,vely)-> None
```

prev First image, 8-bit, single-channel

curr Second image, 8-bit, single-channel

winSize Size of the averaging window used for grouping pixels

velx Horizontal component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

vely Vertical component of the optical flow of the same size as input images, 32-bit floating-point, single-channel

The function computes the flow for every pixel of the first input image using the Lucas and Kanade algorithm [?].

---

cv.CalcOpticalFlowPyrLK

Calculates the optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

```
CalcOpticalFlowPyrLK( prev, curr, prevPyr, currPyr, prevFeatures,
winSize, level, criteria, flags, guesses = None) -> (currFeatures,
status, track_error)
```

prev First frame, at time `t`

curr Second frame, at time `t + dt`

prevPyr Buffer for the pyramid for the first frame. If the pointer is not `NULL` , the buffer must have a sufficient size to store the pyramid from level `1` to level `level` ; the total size of `(image_width+8)*image_height/3` bytes is sufficient

currPyr Similar to `prevPyr`, used for the second frame

prevFeatures Array of points for which the flow needs to be found

currFeatures Array of 2D points containing the calculated new positions of the input features in the second image

winSize Size of the search window of each pyramid level

level Maximal pyramid level number. If `0` , pyramids are not used (single level), if `1` , two levels are used, etc

status Array. Every element of the array is set to `1` if the flow for the corresponding feature has been found, `0` otherwise

track_error Array of double numbers containing the difference between patches around the original and moved points. Optional parameter; can be `NULL`

criteria Specifies when the iteration process of finding the flow for each point on each pyramid level should be stopped

flags Miscellaneous flags:

CV_LKFLOWPyr_A_READY pyramid for the first frame is precalculated before the call

CV_LKFLOWPyr_B_READY pyramid for the second frame is precalculated before the call

guesses optional array of estimated coordinates of features in second frame, with same length as `prevFeatures`

The function implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids [?] . It calculates the coordinates of the feature points on the current video frame given their coordinates on the previous frame. The function finds the coordinates with sub-pixel accuracy.

Both parameters `prevPyr` and `currPyr` comply with the following rules: if the image pointer is 0, the function allocates the buffer internally, calculates the pyramid, and releases the buffer after processing. Otherwise, the function calculates the pyramid and stores it in the buffer unless the flag `CV_LKFLOWPyr_A[B]_READY` is set. The image should be large enough to fit the Gaussian

pyramid data. After the function call both pyramids are calculated and the readiness flag for the corresponding image can be set in the next call (i.e., typically, for all the image pairs except the very first one `CV_LKFLOWPyr_A_READY` is set).

---

cv.CamShift

Finds the object center, size, and orientation.

```
CamShift(prob_image,window,criteria)-> (int, comp, box)
```

`prob_image` Back projection of object histogram (see cv.CalcBackProject)

`window` Initial search window

`criteria` Criteria applied to determine when the window search should be finished

`comp` Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

`box` Circumscribed box for the object.

The function implements the CAMSHIFT object tracking algrorithm [?]. First, it finds an object center using cv.MeanShift and, after that, calculates the object size and orientation. The function returns number of iterations made within cv.MeanShift.

The `CamShiftTracker` class declared in cv.hpp implements the color object tracker that uses the function.

CvKalman

Kalman filter state.

```
typedef struct CvKalman
{
    int MP;                     /* number of measurement vector dimensions */
    int DP;                     /* number of state vector dimensions */
    int CP;                     /* number of control vector dimensions */

    /* backward compatibility fields */
#if 1
    float* PosterState;         /* =state_pre->data.fl */
    float* PriorState;          /* =state_post->data.fl */
```

```
    float* DynamMatr;           /* =transition_matrix->data.fl */
    float* MeasurementMatr;     /* =measurement_matrix->data.fl */
    float* MNCovariance;        /* =measurement_noise_cov->data.fl */
    float* PNCovariance;        /* =process_noise_cov->data.fl */
    float* KalmGainMatr;        /* =gain->data.fl */
    float* PriorErrorCovariance;/* =error_cov_pre->data.fl */
    float* PosterErrorCovariance;/* =error_cov_post->data.fl */
    float* Temp1;               /* temp1->data.fl */
    float* Temp2;               /* temp2->data.fl */
#endif

    CvMat* state_pre;           /* predicted state (x'(k)):
                                   x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* corrected state (x(k)):
                                   x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* state transition matrix (A) */
    CvMat* control_matrix;      /* control matrix (B)
                                   (it is not used if there is no control)*/
    CvMat* measurement_matrix;  /* measurement matrix (H) */
    CvMat* process_noise_cov;   /* process noise covariance matrix (Q) */
    CvMat* measurement_noise_cov; /* measurement noise covariance matrix (R) */
    CvMat* error_cov_pre;       /* priori error estimate covariance matrix (P'(k)):
                                   P'(k)=A*P(k-1)*At + Q*/
    CvMat* gain;                /* Kalman gain matrix (K(k)):
                                   K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* posteriori error estimate covariance matrix (P(k)):
                                   P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

The structure `CvKalman` is used to keep the Kalman filter state. It is created by the cv.CreateKalman function, updated by the cv.KalmanPredict and cv.KalmanCorrect functions and released by the cv.ReleaseKalman function. Normally, the structure is used for the standard Kalman filter (notation and the formulas below are borrowed from the excellent Kalman tutorial [?])

$$x_k = A \cdot x_{k-1} + B \cdot u_k + w_k$$
$$z_k = H \cdot x_k + v_k$$

where:

x_k (x_{k-1})	state of the system at the moment *k* (*k-1*)
z_k	measurement of the system state at the moment *k*
u_k	external control applied at the moment *k*

w_k and v_k are normally-distributed process and measurement noise, respectively:

$$p(w) \sim N(0, Q)$$
$$p(v) \sim N(0, R)$$

that is,
Q process noise covariance matrix, constant or variable,
R measurement noise covariance matrix, constant or variable

In the case of the standard Kalman filter, all of the matrices: A, B, H, Q and R are initialized once after the cv.CvKalman structure is allocated via cv.CreateKalman. However, the same structure and the same functions may be used to simulate the extended Kalman filter by linearizing the extended Kalman filter equation in the current system state neighborhood, in this case A, B, H (and, probably, Q and R) should be updated on every step.

---

cv.CreateKalman

Allocates the Kalman filter structure.

```
CreateKalman(dynam_params, measure_params, control_params=0) -> CvKalman
```

dynam_params dimensionality of the state vector

measure_params dimensionality of the measurement vector

control_params dimensionality of the control vector

The function allocates cv.CvKalman and all its matrices and initializes them somehow.

---

cv.KalmanCorrect

Adjusts the model state.

```
KalmanCorrect(kalman, measurement) -> cvmat
```

kalman Kalman filter object returned by cv.CreateKalman

measurement CvMat containing the measurement vector

```
#define cvKalmanUpdateByMeasurement cvKalmanCorrect
```

The function adjusts the stochastic model state on the basis of the given measurement of the model state:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

where

z_k	given measurement (`mesurement` parameter)
K_k	Kalman "gain" matrix.

The function stores the adjusted state at `kalman->state_post` and returns it on output.

---

cv.KalmanPredict

Estimates the subsequent model state.

```
KalmanPredict(kalman, control=None) -> cvmat
```

```
#define cvKalmanUpdateByTime cvKalmanPredict
```

kalman Kalman filter object returned by cv.CreateKalman

control Control vector u_k, should be NULL iff there is no external control (`control_params` =0)

The function estimates the subsequent stochastic model state by its current state and stores it at `kalman->state_pre`:

$$x'_k = A \cdot x_{k-1} + B \cdot u_k$$
$$P'_k = A \cdot P_{k-1} + A^T + Q$$

where

x'_k	is predicted state `kalman->state_pre`,
x_{k-1}	is corrected state on the previous step `kalman->state_post` (should be initialized somehow in the beginning, zero vector by default),
u_k	is external control (`control` parameter),
P'_k	is priori error covariance matrix `kalman->error_cov_pre`
P_{k-1}	is posteriori error covariance matrix on the previous step `kalman->error_cov_post` (should be initialized somehow in the beginning, identity matrix by default),

The function returns the estimated state.

cv.MeanShift

Finds the object center on back projection.

```
MeanShift(prob_image,window,criteria)-> comp
```

prob_image Back projection of the object histogram (see cv.CalcBackProject)

window Initial search window

criteria Criteria applied to determine when the window search should be finished

comp Resultant structure that contains the converged search window coordinates (`comp->rect` field) and the sum of all of the pixels inside the window (`comp->area` field)

The function iterates to find the object center given its back projection and initial position of search window. The iterations are made until the search window center moves by less than the given value and/or until the function has done the maximum number of iterations. The function returns the number of iterations made.

cv.MultiplyAcc

Adds the product of two input images to the accumulator.

```
MultiplyAcc(image1,image2,acc,mask=NULL)-> None
```

image1 First input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

image2 Second input image, the same format as the first one

acc Accumulator with the same number of channels as input images, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the product of 2 images or their selected regions to the accumulator `acc`:

$$\texttt{acc}(x,y) \leftarrow \texttt{acc}(x,y) + \texttt{image1}(x,y) \cdot \texttt{image2}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cv.RunningAvg

Updates the running average.

```
RunningAvg(image,acc,alpha,mask=NULL)-> None
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

acc Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

alpha Weight of input image

mask Optional operation mask

The function calculates the weighted sum of the input image `image` and the accumulator `acc` so that `acc` becomes a running average of frame sequence:

$$\texttt{acc}(x,y) \leftarrow (1-\alpha) \cdot \texttt{acc}(x,y) + \alpha \cdot \texttt{image}(x,y) \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

where α regulates the update speed (how fast the accumulator forgets about previous frames).

---

cv.SegmentMotion

Segments a whole motion into separate moving parts.

```
SegmentMotion(mhi,seg_mask,storage,timestamp,seg_thresh)-> None
```

mhi Motion history image

seg_mask Image where the mask found should be stored, single-channel, 32-bit floating-point

storage Memory storage that will contain a sequence of motion connected components

timestamp Current time in milliseconds or other units

seg_thresh Segmentation threshold; recommended to be equal to the interval between motion history "steps" or greater

The function finds all of the motion segments and marks them in `seg_mask` with individual values (1,2,...). It also returns a sequence of cv.CvConnectedComp structures, one for each motion component. After that the motion direction for every component can be calculated with cv.CalcGlobalOrientation using the extracted mask of the particular component cv.Cmp.

---

cv.SnakeImage

Changes the contour position to minimize its energy.

```
SnakeImage(image,points,alpha,beta,gamma,coeff_usage,win,criteria,calc_gradient=1)->
None
```

image The source image or external energy field

points Contour points (snake)

alpha Weight[s] of continuity energy, single float or array of `length` floats, one for each contour point

beta Weight[s] of curvature energy, similar to `alpha`

gamma Weight[s] of image energy, similar to `alpha`

coeff_usage Different uses of the previous three parameters:

- **CV_VALUE** indicates that each of `alpha, beta, gamma` is a pointer to a single value to be used for all points;
- **CV_ARRAY** indicates that each of `alpha, beta, gamma` is a pointer to an array of coefficients different for all the points of the snake. All the arrays must have the size equal to the contour size.

win Size of neighborhood of every point used to search the minimum, both `win.width` and `win.height` must be odd

criteria Termination criteria

calc_gradient Gradient flag; if not 0, the function calculates the gradient magnitude for every image pixel and consideres it as the energy field, otherwise the input image itself is considered

The function updates the snake in order to minimize its total energy that is a sum of internal energy that depends on the contour shape (the smoother contour is, the smaller internal energy is) and external energy that depends on the energy field and reaches minimum at the local energy extremums that correspond to the image edges in the case of using an image gradient.

The parameter `criteria.epsilon` is used to define the minimal number of points that must be moved during any iteration to keep the iteration process running.

If at some iteration the number of moved points is less than `criteria.epsilon` or the function performed `criteria.max_iter` iterations, the function terminates.

---

cv.SquareAcc

Adds the square of the source image to the accumulator.

```
SquareAcc(image,sqsum,mask=NULL)-> None
```

image Input image, 1- or 3-channel, 8-bit or 32-bit floating point (each channel of multi-channel image is processed independently)

sqsum Accumulator with the same number of channels as input image, 32-bit or 64-bit floating-point

mask Optional operation mask

The function adds the input image `image` or its selected region, raised to power 2, to the accumulator `sqsum`:

$$\texttt{sqsum}(x,y) \leftarrow \texttt{sqsum}(x,y) + \texttt{image}(x,y)^2 \quad \text{if} \quad \texttt{mask}(x,y) \neq 0$$

---

cv.UpdateMotionHistory

Updates the motion history image by a moving silhouette.

```
UpdateMotionHistory(silhouette,mhi,timestamp,duration)-> None
```

silhouette Silhouette mask that has non-zero pixels where the motion occurs

mhi Motion history image, that is updated by the function (single-channel, 32-bit floating-point)

timestamp Current time in milliseconds or other units

duration Maximal duration of the motion track in the same units as `timestamp`

The function updates the motion history image as following:

$$\texttt{mhi}(x,y) = \begin{cases} \texttt{timestamp} & \text{if } \texttt{silhouette}(x,y) \neq 0 \\ 0 & \text{if } \texttt{silhouette}(x,y) = 0 \text{ and } \texttt{mhi} < (\texttt{timestamp} - \texttt{duration}) \\ \texttt{mhi}(x,y) & \text{otherwise} \end{cases}$$

That is, MHI pixels where motion occurs are set to the current timestamp, while the pixels where motion happened far ago are cleared.

13.7 Structural Analysis and Shape Descriptors

---

cv.ApproxChains

Approximates Freeman chain(s) with a polygonal curve.

```
ApproxChains(src_seq,storage,method=CV_CHAIN_APPROX_SIMPLE,parameter=0,minimal_perime
chains
```

src_seq Pointer to the chain that can refer to other chains

storage Storage location for the resulting polylines

method Approximation method (see the description of the function cv.FindContours)

parameter Method parameter (not used now)

minimal_perimeter Approximates only those contours whose perimeters are not less than `minimal_perimeter`. Other chains are removed from the resulting structure

recursive If not 0, the function approximates all chains that access can be obtained to from `src_seq` by using the `h_next` or `v_next links`. If 0, the single chain is approximated

This is a stand-alone approximation routine. The function `cvApproxChains` works exactly in the same way as cv.FindContours with the corresponding approximation flag. The function returns pointer to the first resultant contour. Other approximated contours, if any, can be accessed via the `v_next` or `h_next` fields of the returned structure.

---

cv.ApproxPoly

Approximates polygonal curve(s) with the specified precision.

```
ApproxPoly(src_seq, storage, method, parameter=0, parameter2=0) ->
sequence
```

src_seq Sequence of an array of points

storage Container for the approximated contours. If it is NULL, the input sequences' storage is used

method Approximation method; only `CV_POLY_APPROX_DP` is supported, that corresponds to the Douglas-Peucker algorithm

parameter Method-specific parameter; in the case of `CV_POLY_APPROX_DP` it is a desired approximation accuracy

parameter2 If case if `src_seq` is a sequence, the parameter determines whether the single sequence should be approximated or all sequences on the same level or below `src_seq` (see cv.FindContours for description of hierarchical contour structures). If `src_seq` is an array CvMat* of points, the parameter specifies whether the curve is closed (`parameter2`!=0) or not (`parameter2` =0)

The function approximates one or more curves and returns the approximation result[s]. In the case of multiple curves, the resultant tree will have the same structure as the input one (1:1 correspondence).

---

cv.ArcLength

Calculates the contour perimeter or the curve length.

```
ArcLength(curve,slice=CV_WHOLE_SEQ,isClosed=-1)-> double
```

curve Sequence or array of the curve points

slice Starting and ending points of the curve, by default, the whole curve length is calculated

isClosed Indicates whether the curve is closed or not. There are 3 cases:

- `isClosed` $= 0$ the curve is assumed to be unclosed.
- `isClosed` > 0 the curve is assumed to be closed.
- `isClosed` < 0 if curve is sequence, the flag `CV_SEQ_FLAG_CLOSED` of `((CvSeq*)curve)->flags` is checked to determine if the curve is closed or not, otherwise (curve is represented by array (CvMat*) of points) it is assumed to be unclosed.

The function calculates the length or curve as the sum of lengths of segments between subsequent points

---

cv.BoundingRect

Calculates the up-right bounding rectangle of a point set.

```
BoundingRect(points,update=0)-> CvRect
```

points 2D point set, either a sequence or vector (`CvMat`) of points

update The update flag. See below.

The function returns the up-right bounding rectangle for a 2d point set. Here is the list of possible combination of the flag values and type of `points`:

update	points	action
0	`CvContour`	the bounding rectangle is not calculated, but it is taken from `rect` field of the contour header.
1	`CvContour`	the bounding rectangle is calculated and written to `rect` field of the contour header.
0	`CvSeq` or `CvMat`	the bounding rectangle is calculated and returned.
1	`CvSeq` or `CvMat`	runtime error is raised.

cv.BoxPoints

Finds the box vertices.

```
BoxPoints(box)-> points
```

box Box

points Array of vertices

The function calculates the vertices of the input 2d box. Here is the function code:

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    float a = (float)cos(box.angle)*0.5f;
    float b = (float)sin(box.angle)*0.5f;

    pt[0].x = box.center.x - a*box.size.height - b*box.size.width;
    pt[0].y = box.center.y + b*box.size.height - a*box.size.width;
    pt[1].x = box.center.x + a*box.size.height - b*box.size.width;
    pt[1].y = box.center.y - b*box.size.height - a*box.size.width;
    pt[2].x = 2*box.center.x - pt[0].x;
    pt[2].y = 2*box.center.y - pt[0].y;
    pt[3].x = 2*box.center.x - pt[1].x;
    pt[3].y = 2*box.center.y - pt[1].y;
}
```

cv.CalcPGH

Calculates a pair-wise geometrical histogram for a contour.

```
CalcPGH(contour,hist)-> None
```

contour Input contour. Currently, only integer point coordinates are allowed

hist Calculated histogram; must be two-dimensional

The function calculates a 2D pair-wise geometrical histogram (PGH), described in cv.livarinen97 for the contour. The algorithm considers every pair of contour edges. The angle between the edges and the minimum/maximum distances are determined for every pair. To do this each of the edges in turn is taken as the base, while the function loops through all the other edges. When the base edge and any other edge are considered, the minimum and maximum distances from the points on the non-base edge and line of the base edge are selected. The angle between the edges defines the row of the histogram in which all the bins that correspond to the distance between the calculated minimum and maximum distances are incremented (that is, the histogram is transposed relatively to the cv.livarninen97 definition). The histogram can be used for contour matching.

---

cv.CalcEMD2

Computes the ”minimal work” distance between two weighted point configurations.

```
CalcEMD2(signature1, signature2, distance_type, distance_func = None,
cost_matrix=None, flow=None, lower_bound=None, userdata = None) -> float
```

signature1 First signature, a $\texttt{size1} \times \texttt{dims} + 1$ floating-point matrix. Each row stores the point weight followed by the point coordinates. The matrix is allowed to have a single column (weights only) if the user-defined cost matrix is used

signature2 Second signature of the same format as `signature1`, though the number of rows may be different. The total weights may be different, in this case an extra ”dummy” point is added to either `signature1` or `signature2`

distance_type Metrics used; `CV_DIST_L1`, `CV_DIST_L2`, and `CV_DIST_C` stand for one of the standard metrics; `CV_DIST_USER` means that a user-defined function `distance_func` or pre-calculated `cost_matrix` is used

distance_func The user-defined distance function. It takes coordinates of two points and returns the distance between the points

cost_matrix The user-defined $\texttt{size1} \times \texttt{size2}$ cost matrix. At least one of `cost_matrix` and `distance_func` must be NULL. Also, if a cost matrix is used, lower boundary (see below) can not be calculated, because it needs a metric function

flow The resultant $\texttt{size1} \times \texttt{size2}$ flow matrix: $\texttt{flow}_{i,j}$ is a flow from i th point of `signature1` to j th point of `signature2`

lower_bound Optional input/output parameter: lower boundary of distance between the two signatures that is a distance between mass centers. The lower boundary may not be calculated if the user-defined cost matrix is used, the total weights of point configurations are not equal, or if the signatures consist of weights only (i.e. the signature matrices have a single column). The user **must** initialize `*lower_bound`. If the calculated distance between mass centers is greater or equal to `*lower_bound` (it means that the signatures are far enough) the function does not calculate EMD. In any case `*lower_bound` is set to the calculated distance between mass centers on return. Thus, if user wants to calculate both distance between mass centers and EMD, `*lower_bound` should be set to 0

userdata Pointer to optional data that is passed into the user-defined distance function

```
typedef float (*CvDistanceFunction)(const float* f1, const float* f2, void* userdata);
```

The function computes the earth mover distance and/or a lower boundary of the distance between the two weighted point configurations. One of the applications described in cv.RubnerSept98 is multi-dimensional histogram comparison for image retrieval. EMD is a a transportation problem that is solved using some modification of a simplex algorithm, thus the complexity is exponential in the worst case, though, on average it is much faster. In the case of a real metric the lower boundary can be calculated even faster (using linear-time algorithm) and it can be used to determine roughly whether the two signatures are far enough so that they cannot relate to the same object.

---

cv.CheckContourConvexity

Tests contour convexity.

```
CheckContourConvexity(contour)-> int
```

contour Tested contour (sequence or array of points)

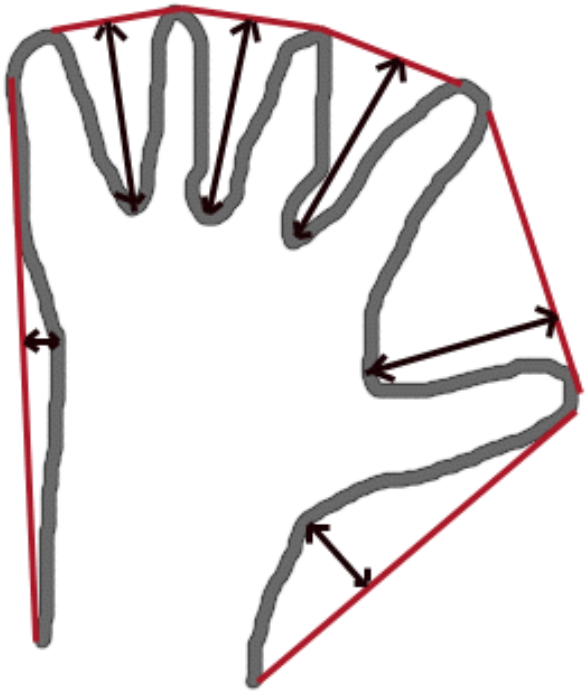
The function tests whether the input contour is convex or not. The contour must be simple, without self-intersections.

CvConvexityDefect

Structure describing a single contour convexity defect.

```
typedef struct CvConvexityDefect
{
    CvPoint* start; /* point of the contour where the defect begins */
```

```
    CvPoint* end; /* point of the contour where the defect ends */
    CvPoint* depth_point; /* the farthest from the convex hull point within the defect */
    float depth; /* distance between the farthest point and the convex hull */
} CvConvexityDefect;
```



cv.ContourArea

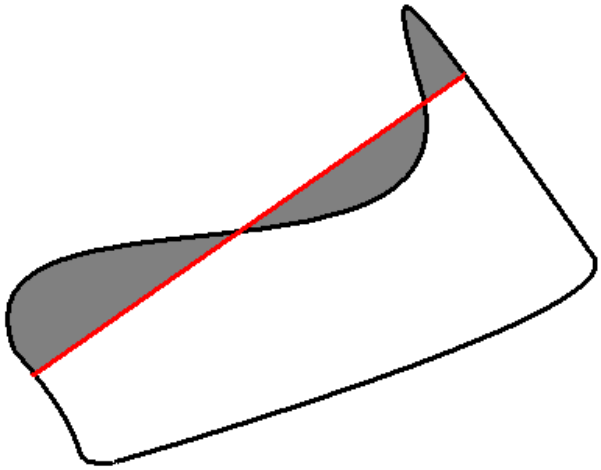
Calculates the area of a whole contour or a contour section.

```
ContourArea(contour,slice=CV_WHOLE_SEQ)-> double
```

contour Contour (sequence or array of vertices)

slice Starting and ending points of the contour section of interest, by default, the area of the whole contour is calculated

The function calculates the area of a whole contour or a contour section. In the latter case the total area bounded by the contour arc and the chord connecting the 2 selected points is calculated as shown on the picture below:



Orientation of the contour affects the area sign, thus the function may return a *negative* result. Use the `fabs()` function from C runtime to get the absolute value of the area.

---

cv.ContourFromContourTree

Restores a contour from the tree.

```
ContourFromContourTree(tree,storage,criteria)-> contour
```

`tree` Contour tree

`storage` Container for the reconstructed contour

`criteria` Criteria, where to stop reconstruction

The function restores the contour from its binary tree representation. The parameter `criteria` determines the accuracy and/or the number of tree levels used for reconstruction, so it is possible to build an approximated contour. The function returns the reconstructed contour.

---

cv.ConvexHull2

Finds the convex hull of a point set.

```
ConvexHull2(points,storage,orientation=CV_CLOCKWISE,return_points=0)->
convex_hull
```

points Sequence or array of 2D points with 32-bit integer or floating-point coordinates

storage The destination array (CvMat*) or memory storage (CvMemStorage*) that will store the convex hull. If it is an array, it should be 1d and have the same number of elements as the input array/sequence. On output the header is modified as to truncate the array down to the hull size. If `storage` is NULL then the convex hull will be stored in the same storage as the input sequence

orientation Desired orientation of convex hull: `CV_CLOCKWISE` or `CV_COUNTER_CLOCKWISE`

return_points If non-zero, the points themselves will be stored in the hull instead of indices if `storage` is an array, or pointers if `storage` is memory storage

The function finds the convex hull of a 2D point set using Sklansky's algorithm. If `storage` is memory storage, the function creates a sequence containing the hull points or pointers to them, depending on `return_points` value and returns the sequence on output. If `storage` is a CvMat, the function returns NULL.

```
#include "cv.h"
#include "highgui.h"
#include <stdlib.h>

#define ARRAY  0 /* switch between array/sequence method by replacing 0<=>1 */

void main( int argc, char** argv )
{
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    cvNamedWindow( "hull", 1 );

#if !ARRAY
        CvMemStorage* storage = cvCreateMemStorage();
#endif

    for(;;)
    {
        int i, count = rand()%100 + 1, hullcount;
        CvPoint pt0;
#if !ARRAY
        CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2,
                                    sizeof(CvContour),
                                    sizeof(CvPoint),
                                    storage );
        CvSeq* hull;

        for( i = 0; i < count; i++ )
```

```
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            cvSeqPush( ptseq, &pt0 );
        }
        hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
        hullcount = hull->total;
#else
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
        int* hull = (int*)malloc( count * sizeof(hull[0]));
        CvMat point_mat = cvMat( 1, count, CV_32SC2, points );
        CvMat hull_mat = cvMat( 1, count, CV_32SC1, hull );

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            points[i] = pt0;
        }
        cvConvexHull2( &point_mat, &hull_mat, CV_CLOCKWISE, 0 );
        hullcount = hull_mat.cols;
#endif
        cvZero( img );
        for( i = 0; i < count; i++ )
        {
#if !ARRAY
            pt0 = *CV_GET_SEQ_ELEM( CvPoint, ptseq, i );
#else
            pt0 = points[i];
#endif
            cvCircle( img, pt0, 2, CV_RGB( 255, 0, 0 ), CV_FILLED );
        }

#if !ARRAY
        pt0 = **CV_GET_SEQ_ELEM( CvPoint*, hull, hullcount - 1 );
#else
        pt0 = points[hull[hullcount-1]];
#endif

        for( i = 0; i < hullcount; i++ )
        {
#if !ARRAY
            CvPoint pt = **CV_GET_SEQ_ELEM( CvPoint*, hull, i );
#else
            CvPoint pt = points[hull[i]];
```

```
#endif
            cvLine( img, pt0, pt, CV_RGB( 0, 255, 0 ));
            pt0 = pt;
        }

        cvShowImage( "hull", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;

#if !ARRAY
        cvClearMemStorage( storage );
#else
        free( points );
        free( hull );
#endif
    }
}
```

cv.ConvexityDefects

Finds the convexity defects of a contour.

```
ConvexityDefects(contour,convexhull,storage)-> convexity_defects
```

contour Input contour

convexhull Convex hull obtained using cv.ConvexHull2 that should contain pointers or indices to the contour points, not the hull points themselves (the `return_points` parameter in cv.ConvexHull2 should be 0)

storage Container for the output sequence of convexity defects. If it is NULL, the contour or hull (in that order) storage is used

The function finds all convexity defects of the input contour and returns a sequence of the CvConvexityDefect structures.

cv.CreateContourTree

Creates a hierarchical representation of a contour.

```
CreateContourTree(contour,storage,threshold)-> contour_tree
```

contour Input contour

storage Container for output tree

threshold Approximation accuracy

The function creates a binary tree representation for the input `contour` and returns the pointer to its root. If the parameter `threshold` is less than or equal to 0, the function creates a full binary tree representation. If the threshold is greater than 0, the function creates a representation with the precision `threshold`: if the vertices with the interceptive area of its base line are less than `threshold`, the tree should not be built any further. The function returns the created tree.

---

cv.FindContours

Finds the contours in a binary image.

```
FindContours(image, storage, mode=CV_RETR_LIST, method=CV_CHAIN_APPROX_SIMPLE,
offset=(0,0)) -> cvseq
```

image The source, an 8-bit single channel image. Non-zero pixels are treated as 1's, zero pixels remain 0's - the image is treated as `binary`. To get such a binary image from grayscale, one may use cv.Threshold, cv.AdaptiveThreshold or cv.Canny. The function modifies the source image's content

storage Container of the retrieved contours

mode Retrieval mode

- **CV_RETR_EXTERNAL** retrives only the extreme outer contours
- **CV_RETR_LIST** retrieves all of the contours and puts them in the list
- **CV_RETR_CCOMP** retrieves all of the contours and organizes them into a two-level hierarchy: on the top level are the external boundaries of the components, on the second level are the boundaries of the holes
- **CV_RETR_TREE** retrieves all of the contours and reconstructs the full hierarchy of nested contours

method Approximation method (for all the modes, except CV_LINK_RUNS, which uses built-in approximation)

- **CV_CHAIN_CODE** outputs contours in the Freeman chain code. All other methods output polygons (sequences of vertices)
- **CV_CHAIN_APPROX_NONE** translates all of the points from the chain code into points
- **CV_CHAIN_APPROX_SIMPLE** compresses horizontal, vertical, and diagonal segments and leaves only their end points
- **CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS** applies one of the flavors of the Teh-Chin chain approximation algorithm.
- **CV_LINK_RUNS** uses a completely different contour retrieval algorithm by linking horizontal segments of 1's. Only the CV_RETR_LIST retrieval mode can be used with this method.

offset Offset, by which every contour point is shifted. This is useful if the contours are extracted from the image ROI and then they should be analyzed in the whole image context

The function retrieves contours from the binary image and returns the number of retrieved contours. The pointer first_contour is filled by the function. It will contain a pointer to the first outermost contour or NULL if no contours are detected (if the image is completely black). Other contours may be reached from first_contour using the h_next and v_next links. The sample in the cv.DrawContours discussion shows how to use contours for connected component detection. Contours can be also used for shape analysis and object recognition - see squares.c in the OpenCV sample directory.

---

cv.FitEllipse2

Fits an ellipse around a set of 2D points.

```
FitEllipse2(points)-> Box2D
```

points Sequence or array of points

The function calculates the ellipse that fits best (in least-squares sense) around a set of 2D points. The meaning of the returned structure fields is similar to those in cv.Ellipse except that size stores the full lengths of the ellipse axises, not half-lengths.

cv.FitLine

Fits a line to a 2D or 3D point set.

```
FitLine(points, dist_type, param, reps, aeps) -> line
```

points Sequence or array of 2D or 3D points with 32-bit integer or floating-point coordinates

dist_type The distance used for fitting (see the discussion)

param Numerical parameter (C) for some types of distances, if 0 then some optimal value is chosen

reps Sufficient accuracy for the radius (distance between the coordinate origin and the line). 0.01 is a good default value.

aeps Sufficient accuracy for the angle. 0.01 is a good default value.

line The output line parameters. In the case of a 2d fitting, it is a tuple of 4 floats `(vx, vy, x0, y0)` where `(vx, vy)` is a normalized vector collinear to the line and `(x0, y0)` is some point on the line. in the case of a 3D fitting it is a tuple of 6 floats `(vx, vy, vz, x0, y0, z0)` where `(vx, vy, vz)` is a normalized vector collinear to the line and `(x0, y0, z0)` is some point on the line

The function fits a line to a 2D or 3D point set by minimizing $\sum_i \rho(r_i)$ where r_i is the distance between the i th point and the line and $\rho(r)$ is a distance function, one of:

dist_type=CV_DIST_L2

$$\rho(r) = r^2/2 \quad \text{(the simplest and the fastest least-squares method)}$$

dist_type=CV_DIST_L1

$$\rho(r) = r$$

dist_type=CV_DIST_L12

$$\rho(r) = 2 \cdot (\sqrt{1 + \frac{r^2}{2}} - 1)$$

dist_type=CV_DIST_FAIR

$$\rho\left(r\right) = C^2 \cdot \left(\frac{r}{C} - \log\left(1 + \frac{r}{C}\right)\right) \quad \text{where} \quad C = 1.3998$$

dist_type=CV_DIST_WELSCH

$$\rho(r) = \frac{C^2}{2} \cdot \left(1 - \exp\left(-\left(\frac{r}{C}\right)^2\right)\right) \quad \text{where} \quad C = 2.9846$$

dist_type=CV_DIST_HUBER

$$\rho(r) = \begin{cases} r^2/2 & \text{if } r < C \\ C \cdot (r - C/2) & \text{otherwise} \end{cases} \quad \text{where} \quad C = 1.345$$

---

cv.GetCentralMoment

Retrieves the central moment from the moment state structure.

```
GetCentralMoment(moments, x_order, y_order) -> double
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, x_order >= 0

y_order y order of the retrieved moment, y_order >= 0 and x_order + y_order <= 3

The function retrieves the central moment, which in the case of image moments is defined as:

$$\mu_{x\_order,\, y\_order} = \sum_{x,y} (I(x,y) \cdot (x - x_c)^{x\_order} \cdot (y - y_c)^{y\_order})$$

where x_c, y_c are the coordinates of the gravity center:

$$x_c = \frac{M_{10}}{M_{00}}, y_c = \frac{M_{01}}{M_{00}}$$

---

cv.GetHuMoments

Calculates the seven Hu invariants.

```
GetHuMoments(moments) -> hu
```

moments The input moments, computed with cv.Moments

hu The output Hu invariants

The function calculates the seven Hu invariants, see http://en.wikipedia.org/wiki/Image_moment, that are defined as:

$$
\begin{aligned}
hu_1 &= \eta_{20} + \eta_{02} \\
hu_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
hu_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
hu_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
hu_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
hu_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
hu_7 &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned}
$$

where η_{ji} denote the normalized central moments.

These values are proved to be invariant to the image scale, rotation, and reflection except the seventh one, whose sign is changed by reflection. Of course, this invariance was proved with the assumption of infinite image resolution. In case of a raster images the computed Hu invariants for the original and transformed images will be a bit different.

---

cv.GetNormalizedCentralMoment

Retrieves the normalized central moment from the moment state structure.

```
GetNormalizedCentralMoment(moments, x_order, y_order) -> double
```

moments Pointer to the moment state structure

x_order x order of the retrieved moment, `x_order` $>= 0$

y_order y order of the retrieved moment, `y_order` $>= 0$ and `x_order` + `y_order` $<= 3$

The function retrieves the normalized central moment:

$$\eta_{x\_order, y\_order} = \frac{\mu_{x\_order, y\_order}}{M_{00}^{(y\_order + x\_order)/2+1}}$$

cv.GetSpatialMoment

Retrieves the spatial moment from the moment state structure.

```
GetSpatialMoment(moments, x_order, y_order) -> double
```

moments The moment state, calculated by cv.Moments

x_order x order of the retrieved moment, x_order $>= 0$

y_order y order of the retrieved moment, y_order $>= 0$ and x_order + y_order $<= 3$

The function retrieves the spatial moment, which in the case of image moments is defined as:

$$M_{x\_order,\, y\_order} = \sum_{x,y} (I(x,y) \cdot x^{x\_order} \cdot y^{y\_order})$$

where $I(x,y)$ is the intensity of the pixel (x,y).

cv.MatchContourTrees

Compares two contours using their tree representations.

```
MatchContourTrees(tree1,tree2,method,threshold)-> double
```

tree1 First contour tree

tree2 Second contour tree

method Similarity measure, only CV_CONTOUR_TREES_MATCH_I1 is supported

threshold Similarity threshold

The function calculates the value of the matching measure for two contour trees. The similarity measure is calculated level by level from the binary tree roots. If at a certain level the difference between contours becomes less than threshold, the reconstruction process is interrupted and the current difference is returned.

cv.MatchShapes

Compares two shapes.

```
MatchShapes(object1,object2,method,parameter=0)-> None
```

object1 First contour or grayscale image

object2 Second contour or grayscale image

method Comparison method; CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3

parameter Method-specific parameter (is not used now)

The function compares two shapes. The 3 implemented methods all use Hu moments (see cv.GetHuMoments) (A is object1, B is object2):

method=CV_CONTOUR_MATCH_I1

$$I_1(A,B) = \sum_{i=1...7} \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

method=CV_CONTOUR_MATCH_I2

$$I_2(A,B) = \sum_{i=1...7} \left| m_i^A - m_i^B \right|$$

method=CV_CONTOUR_MATCH_I3

$$I_3(A,B) = \sum_{i=1...7} \frac{\left| m_i^A - m_i^B \right|}{\left| m_i^A \right|}$$

where

$$m_i^A = sign(h_i^A) \cdot \log h_i^A m_i^B = sign(h_i^B) \cdot \log h_i^B$$

and h_i^A, h_i^B are the Hu moments of A and B respectively.

cv.MinAreaRect2

Finds the circumscribed rectangle of minimal area for a given 2D point set.

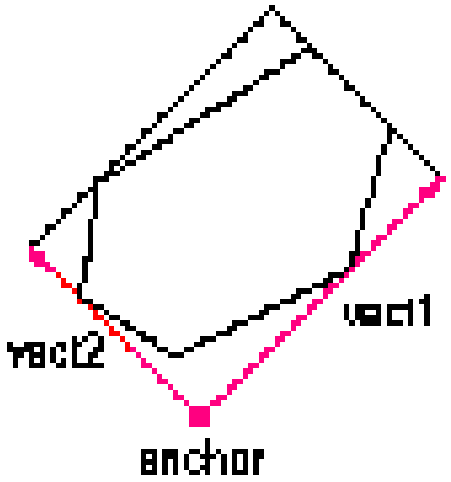
```
MinAreaRect2(points,storage=NULL)-> CvBox2D
```

points Sequence or array of points

storage Optional temporary memory storage

The function finds a circumscribed rectangle of the minimal area for a 2D point set by building a convex hull for the set and applying the rotating calipers technique to the hull.

Picture. Minimal-area bounding rectangle for contour



---

cv.MinEnclosingCircle

Finds the circumscribed circle of minimal area for a given 2D point set.

```
MinEnclosingCircle(points)-> (int,center,radius)
```

points Sequence or array of 2D points

center Output parameter; the center of the enclosing circle

radius Output parameter; the radius of the enclosing circle

The function finds the minimal circumscribed circle for a 2D point set using an iterative algorithm. It returns nonzero if the resultant circle contains all the input points and zero otherwise (i.e. the algorithm failed).

---

cv.Moments

Calculates all of the moments up to the third order of a polygon or rasterized shape.

```
Moments(arr, binary) -> moments
```

arr Image (1-channel or 3-channel with COI set) or polygon (CvSeq of points or a vector of points)

moments Pointer to returned moment's state structure

binary (For images only) If the flag is non-zero, all of the zero pixel values are treated as zeroes, and all of the others are treated as 1's

The function calculates spatial and central moments up to the third order and writes them to `moments`. The moments may then be used then to calculate the gravity center of the shape, its area, main axises and various shape characeteristics including 7 Hu invariants.

---

cv.PointPolygonTest

Point in contour test.

```
PointPolygonTest(contour,pt,measure_dist)-> double
```

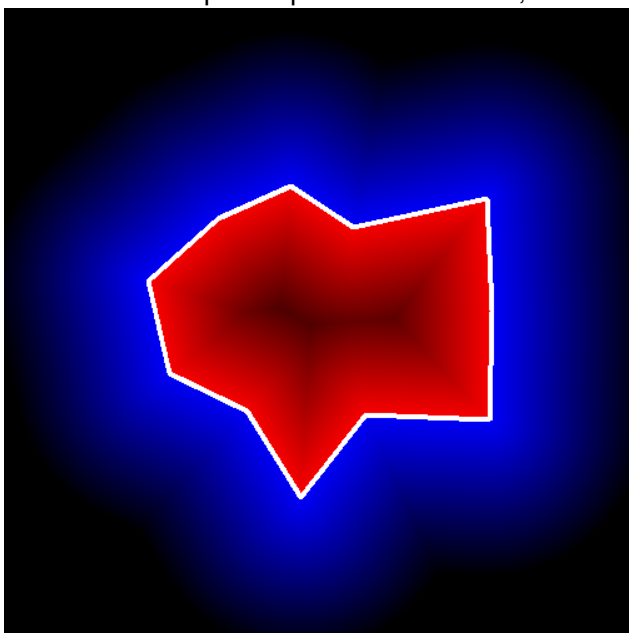
contour Input contour

pt The point tested against the contour

measure_dist If it is non-zero, the function estimates the distance from the point to the nearest contour edge

The function determines whether the point is inside a contour, outside, or lies on an edge (or coinsides with a vertex). It returns positive, negative or zero value, correspondingly. When `measure_dist` $= 0$, the return value is +1, -1 and 0, respectively. When `measure_dist` $\neq 0$, it is a signed distance between the point and the nearest contour edge.

Here is the sample output of the function, where each image pixel is tested against the contour.



13.8 Planar Subdivisions

CvSubdiv2D

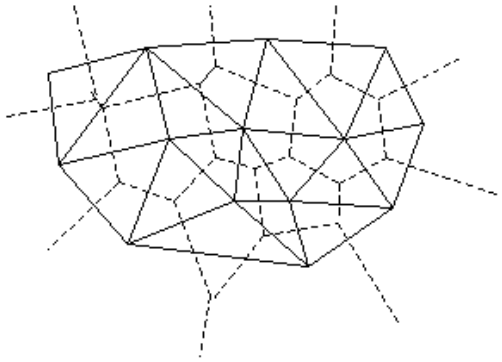
Planar subdivision.

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
```

```
CvSubdiv2D;
```

Planar subdivision is the subdivision of a plane into a set of non-overlapped regions (facets) that cover the whole plane. The above structure describes a subdivision built on a 2d point set, where the points are linked together and form a planar graph, which, together with a few edges connecting the exterior subdivision points (namely, convex hull points) with infinity, subdivides a plane into facets by its edges.

For every subdivision there exists a dual subdivision in which facets and points (subdivision vertices) swap their roles, that is, a facet is treated as a vertex (called a virtual point below) of the dual subdivision and the original subdivision vertices become facets. On the picture below original subdivision is marked with solid lines and dual subdivision with dotted lines.



OpenCV subdivides a plane into triangles using Delaunay's algorithm. Subdivision is built iteratively starting from a dummy triangle that includes all the subdivision points for sure. In this case the dual subdivision is a Voronoi diagram of the input 2d point set. The subdivisions can be used for the 3d piece-wise transformation of a plane, morphing, fast location of points on the plane, building special graphs (such as NNG,RNG) and so forth.

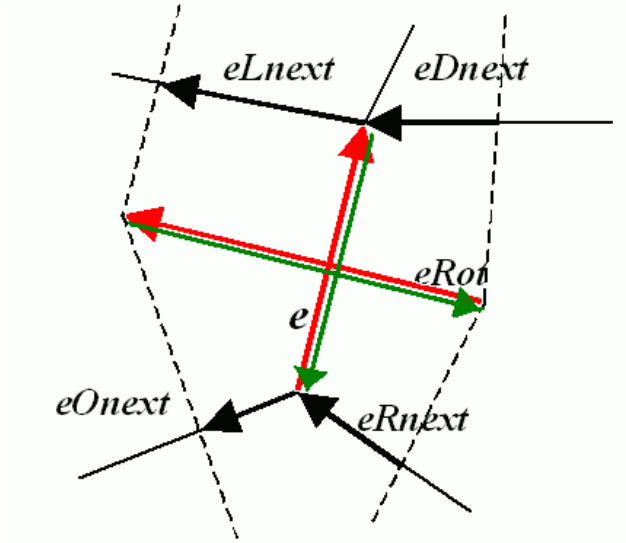
CvQuadEdge2D

Quad-edge of planar subdivision.

```
/* one of edges within quad-edge, lower 2 bits is index (0..3)
   and upper bits are quad-edge pointer */
typedef long CvSubdiv2DEdge;

/* quad-edge structure fields */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];
```

```
typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```

Quad-edge is a basic element of subdivision containing four edges (e, eRot, reversed e and reversed eRot):



CvSubdiv2DPoint

Point of original or dual subdivision.

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int            flags;      \
    CvSubdiv2DEdge first;      \
    CvPoint2D32f   pt;         \
    int id;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

id This integer can be used to index auxillary data associated with each vertex of the planar subdivision

---

cv.CalcSubdivVoronoi2D

Calculates the coordinates of Voronoi diagram cells.

```
CalcSubdivVoronoi2D(subdiv)-> None
```

subdiv Delaunay subdivision, in which all the points are already added

The function calculates the coordinates of virtual points. All virtual points corresponding to some vertex of the original subdivision form (when connected together) a boundary of the Voronoi cell at that point.

---

cv.ClearSubdivVoronoi2D

Removes all virtual points.

```
ClearSubdivVoronoi2D(subdiv)-> None
```

subdiv Delaunay subdivision

The function removes all of the virtual points. It is called internally in cv.CalcSubdivVoronoi2D if the subdivision was modified after previous call to the function.

---

cv.CreateSubdivDelaunay2D

Creates an empty Delaunay triangulation.

```
CreateSubdivDelaunay2D(rect,storage)-> delaunay_triangulation
```

rect Rectangle that includes all of the 2d points that are to be added to the subdivision

storage Container for subdivision

The function creates an empty Delaunay subdivision, where 2d points can be added using the function cv.SubdivDelaunay2DInsert. All of the points to be added must be within the specified rectangle, otherwise a runtime error will be raised.

Note that the triangulation is a single large triangle that covers the given rectangle. Hence the three vertices of this triangle are outside the rectangle `rect`.

---

cv.FindNearestPoint2D

Finds the closest subdivision vertex to the given point.

```
FindNearestPoint2D(subdiv,pt)-> point
```

subdiv Delaunay or another subdivision

pt Input point

The function is another function that locates the input point within the subdivision. It finds the subdivision vertex that is the closest to the input point. It is not necessarily one of vertices of the facet containing the input point, though the facet (located using cv.Subdiv2DLocate) is used as a starting point. The function returns a pointer to the found subdivision vertex.

---

cv.Subdiv2DEdgeDst

Returns the edge destination.

```
Subdiv2DEdgeDst(edge)-> point
```

edge Subdivision edge (not a quad-edge)

The function returns the edge destination. The returned pointer may be NULL if the edge is from dual subdivision and the virtual point coordinates are not calculated yet. The virtual points can be calculated using the function cv.CalcSubdivVoronoi2D.

---

cv.Subdiv2DGetEdge

Returns one of the edges related to the given edge.

```
Subdiv2DGetEdge(edge,type)-> CvSubdiv2DEdge
```

```
#define cvSubdiv2DNextEdge( edge ) cvSubdiv2DGetEdge( edge, CV_NEXT_AROUND_ORG )
```

edge Subdivision edge (not a quad-edge)

type Specifies which of the related edges to return, one of the following:

- **CV_NEXT_AROUND_ORG** next around the edge origin (`eOnext` on the picture above if `e` is the input edge)
- **CV_NEXT_AROUND_DST** next around the edge vertex (`eDnext`)
- **CV_PREV_AROUND_ORG** previous around the edge origin (reversed `eRnext`)
- **CV_PREV_AROUND_DST** previous around the edge destination (reversed `eLnext`)
- **CV_NEXT_AROUND_LEFT** next around the left facet (`eLnext`)
- **CV_NEXT_AROUND_RIGHT** next around the right facet (`eRnext`)
- **CV_PREV_AROUND_LEFT** previous around the left facet (reversed `eOnext`)
- **CV_PREV_AROUND_RIGHT** previous around the right facet (reversed `eDnext`)

The function returns one of the edges related to the input edge.

---

cv.Subdiv2DLocate

Returns the location of a point within a Delaunay triangulation.

```
Subdiv2DLocate(subdiv, pt) -> (loc, where)
```

subdiv Delaunay or another subdivision

pt The point to locate

loc The location of the point within the triangulation

where The edge or vertex. See below.

The function locates the input point within the subdivision. There are 5 cases:

- The point falls into some facet. `loc` is `CV_PTLOC_INSIDE` and `where` is one of edges of the facet.
- The point falls onto the edge. `loc` is `CV_PTLOC_ON_EDGE` and `where` is the edge.
- The point coincides with one of the subdivision vertices. `loc` is `CV_PTLOC_VERTEX` and `where` is the vertex.
- The point is outside the subdivsion reference rectangle. `loc` is `CV_PTLOC_OUTSIDE_RECT` and `where` is None.
- One of input arguments is invalid. The function raises an exception.

---

cv.Subdiv2DRotateEdge

Returns another edge of the same quad-edge.

```
Subdiv2DRotateEdge(edge,rotate)-> CvSubdiv2DEdge
```

edge Subdivision edge (not a quad-edge)

rotate Specifies which of the edges of the same quad-edge as the input one to return, one of the following:

- **0** the input edge (`e` on the picture above if `e` is the input edge)
- **1** the rotated edge (`eRot`)
- **2** the reversed edge (reversed `e` (in green))
- **3** the reversed rotated edge (reversed `eRot` (in green))

The function returns one of the edges of the same quad-edge as the input edge.

---

cv.SubdivDelaunay2DInsert

Inserts a single point into a Delaunay triangulation.

```
SubdivDelaunay2DInsert(subdiv,pt)-> point
```

subdiv Delaunay subdivision created by the function cv.CreateSubdivDelaunay2D

pt Inserted point

The function inserts a single point into a subdivision and modifies the subdivision topology appropriately. If a point with the same coordinates exists already, no new point is added. The function returns a pointer to the allocated point. No virtual point coordinates are calculated at this stage.

13.9 Object Detection

---

cv.MatchTemplate

Compares a template against overlapped image regions.

```
MatchTemplate(image,templ,result,method)-> None
```

image Image where the search is running; should be 8-bit or 32-bit floating-point

templ Searched template; must be not greater than the source image and the same data type as the image

result A map of comparison results; single-channel 32-bit floating-point. If `image` is $W \times H$ and `templ` is $w \times h$ then `result` must be $(W - w + 1) \times (H - h + 1)$

method Specifies the way the template must be compared with the image regions (see below)

The function is similar to cv.CalcBackProjectPatch. It slides through `image`, compares the overlapped patches of size $w \times h$ against `templ` using the specified method and stores the comparison results to `result`. Here are the formulas for the different comparison methods one may use (I denotes `image`, T `template`, R `result`). The summation is done over template and/or the image patch: $x' = 0...w - 1, y' = 0...h - 1$

method=CV_TM_SQDIFF

$$R(x, y) = \sum_{x',y'} (T(x', y') - I(x + x', y + y'))^2$$

method=CV_TM_SQDIFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCORR

$$R(x,y) = \sum_{x',y'}(T(x',y') \cdot I(x+x',y+y'))$$

method=CV_TM_CCORR_NORMED

$$R(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

method=CV_TM_CCOEFF

$$R(x,y) = \sum_{x',y'}(T'(x',y') \cdot I(x+x',y+y'))$$

where

$$T'(x',y') = T(x',y') - 1/(w \cdot h) \cdot \sum_{x'',y''} T(x'',y'')$$
$$I'(x+x',y+y') = I(x+x',y+y') - 1/(w \cdot h) \cdot \sum_{x'',y''} I(x+x'',y+y'')$$

method=CV_TM_CCOEFF_NORMED

$$R(x,y) = \frac{\sum_{x',y'}(T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}}$$

After the function finishes the comparison, the best matches can be found as global minimums (`CV_TM_SQDIFF`) or maximums (`CV_TM_CCORR` and `CV_TM_CCOEFF`) using the cv.MinMaxLoc function. In the case of a color image, template summation in the numerator and each sum in the denominator is done over all of the channels (and separate mean values are used for each channel).

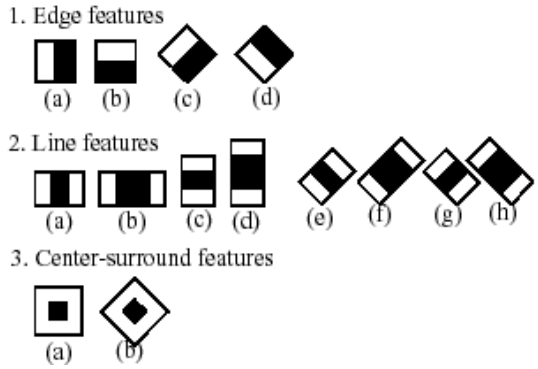
Haar Feature-based Cascade Classifier for Object Detection

The object detector described below has been initially proposed by Paul Viola cv.Viola01 and improved by Rainer Lienhart cv.Lienhart02 . First, a classifier (namely a *cascade of boosted classifiers working with haar-like features*) is trained with a few hundred sample views of a particular

object (i.e., a face or a car), called positive examples, that are scaled to the same size (say, 20x20), and negative examples - arbitrary images of the same size.

After a classifier is trained, it can be applied to a region of interest (of the same size as used during the training) in an input image. The classifier outputs a "1" if the region is likely to show the object (i.e., face/car), and "0" otherwise. To search for the object in the whole image one can move the search window across the image and check every location using the classifier. The classifier is designed so that it can be easily "resized" in order to be able to find the objects of interest at different sizes, which is more efficient than resizing the image itself. So, to find an object of an unknown size in the image the scan procedure should be done several times at different scales.

The word "cascade" in the classifier name means that the resultant classifier consists of several simpler classifiers (*stages*) that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed. The word "boosted" means that the classifiers at every stage of the cascade are complex themselves and they are built out of basic classifiers using one of four different `boosting` techniques (weighted voting). Currently Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost are supported. The basic classifiers are decision-tree classifiers with at least 2 leaves. Haar-like features are the input to the basic classifers, and are calculated as described below. The current algorithm uses the following Haar-like features:



The feature used in a particular classifier is specified by its shape (1a, 2b etc.), position within the region of interest and the scale (this scale is not the same as the scale used at the detection stage, though these two scales are multiplied). For example, in the case of the third line feature (2c) the response is calculated as the difference between the sum of image pixels under the rectangle covering the whole feature (including the two white stripes and the black stripe in the middle) and the sum of the image pixels under the black stripe multiplied by 3 in order to compensate for the differences in the size of areas. The sums of pixel values over a rectangular regions are calculated rapidly using integral images (see below and the cv.Integral description).

A simple demonstration of face detection, which draws a rectangle around each detected face:

```
hc = cv.Load("haarcascade_frontalface_default.xml")
```

```
img = cv.LoadImage("faces.jpg", 0)
faces = cv.HaarDetectObjects(img, hc, cv.CreateMemStorage())
for (x,y,w,h),n in faces:
    cv.Rectangle(img, (x,y), (x+w,y+h), 255)
cv.SaveImage("faces_detected.jpg", img)
```

---

cv.HaarDetectObjects

Detects objects in the image.

```
HaarDetectObjects(image,cascade,storage,scale_factor=1.1,min_neighbors=3,flags=0,mi
detected_objects
```

image Image to detect objects in

cascade Haar classifier cascade in internal representation

storage Memory storage to store the resultant sequence of the object candidate rectangles

scale_factor The factor by which the search window is scaled between the subsequent scans, 1.1 means increasing window by 10%

min_neighbors Minimum number (minus 1) of neighbor rectangles that makes up an object. All the groups of a smaller number of rectangles than `min_neighbors`-1 are rejected. If `min_neighbors` is 0, the function does not any grouping at all and returns all the detected candidate rectangles, which may be useful if the user wants to apply a customized grouping procedure

flags Mode of operation. Currently the only flag that may be specified is `CV_HAAR_DO_CANNY_PRUNING`. If it is set, the function uses Canny edge detector to reject some image regions that contain too few or too much edges and thus can not contain the searched object. The particular threshold values are tuned for face detection and in this case the pruning speeds up the processing

min_size Minimum window size. By default, it is set to the size of samples the classifier has been trained on ($\sim 20 \times 20$ for face detection)

The function finds rectangular regions in the given image that are likely to contain objects the cascade has been trained for and returns those regions as a sequence of rectangles. The function scans the image several times at different scales (see cv.SetImagesForHaarClassifierCascade).

Each time it considers overlapping regions in the image and applies the classifiers to the regions using cv.RunHaarClassifierCascade. It may also apply some heuristics to reduce number of analyzed regions, such as Canny prunning. After it has proceeded and collected the candidate rectangles (regions that passed the classifier cascade), it groups them and returns a sequence of average rectangles for each large enough group. The default parameters (`scale_factor` =1.1, `min_neighbors` =3, `flags` =0) are tuned for accurate yet slow object detection. For a faster operation on real video images the settings are: `scale_factor` =1.2, `min_neighbors` =2, `flags` =`CV_HAAR_DO_CANNY_PRUNING`, `min_size` =*minimum possible face size* (for example, $\sim$ 1/4 to 1/16 of the image area in the case of video conferencing).

13.10 Camera Calibration and 3D Reconstruction

The functions in this section use the so-called pinhole camera model. That is, a scene view is formed by projecting 3D points into the image plane using a perspective transformation.

$$s\ m' = A[R|t]M'$$

or

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where (X, Y, Z) are the coordinates of a 3D point in the world coordinate space, (u, v) are the coordinates of the projection point in pixels. A is called a camera matrix, or a matrix of intrinsic parameters. (cx, cy) is a principal point (that is usually at the image center), and fx, fy are the focal lengths expressed in pixel-related units. Thus, if an image from camera is scaled by some factor, all of these parameters should be scaled (multiplied/divided, respectively) by the same factor. The matrix of intrinsic parameters does not depend on the scene viewed and, once estimated, can be re-used (as long as the focal length is fixed (in case of zoom lens)). The joint rotation-translation matrix $[R|t]$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of still camera. That is, $[R|t]$ translates coordinates of a point (X, Y, Z) to some coordinate system, fixed with respect to the camera. The transformation above is equivalent to the following (when $z \neq 0$):

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Real lenses usually have some distortion, mostly radial distorion and slight tangential distortion. So, the above model is extended as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$
$$x' = x/z$$
$$y' = y/z$$
$$x'' = x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$
$$y'' = y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$
where $r^2 = x'^2 + y'^2$
$$u = f_x * x'' + c_x$$
$$v = f_y * y'' + c_y$$

k_1, k_2, k_3 are radial distortion coefficients, p_1, p_2 are tangential distortion coefficients. Higher-order coefficients are not considered in OpenCV. In the functions below the coefficients are passed or returned as

$$(k_1, k_2, p_1, p_2[, k_3])$$

vector. That is, if the vector contains 4 elements, it means that $k_3 = 0$. The distortion coefficients do not depend on the scene viewed, thus they also belong to the intrinsic camera parameters. *And they remain the same regardless of the captured image resolution.* That is, if, for example, a camera has been calibrated on images of 320×240 resolution, absolutely the same distortion coefficients can be used for images of 640×480 resolution from the same camera (while f_x, f_y, c_x and c_y need to be scaled appropriately).

The functions below use the above model to

- Project 3D points to the image plane given intrinsic and extrinsic parameters
- Compute extrinsic parameters given intrinsic parameters, a few 3D points and their projections.
- Estimate intrinsic and extrinsic camera parameters from several views of a known calibration pattern (i.e. every view is described by several 3D-2D point correspodences).

- Estimate the relative position and orientation of the stereo camera ”heads” and compute the *rectification* transformation that makes the camera optical axes parallel.

---

cv.CalcImageHomography

Calculates the homography matrix for an oblong planar object (e.g. arm).

```
CalcImageHomography(line,center)-> (intrinsic,homography)
```

line the main object axis direction (vector (dx,dy,dz))

center object center ((cx,cy,cz))

intrinsic intrinsic camera parameters (3x3 matrix)

homography output homography matrix (3x3)

The function calculates the homography matrix for the initial image transformation from image plane to the plane, defined by a 3D oblong object line (See __Figure 6-10__ in the OpenCV Guide 3D Reconstruction Chapter).

CalibrateCamera2

calibrateCamera Finds the camera intrinsic and extrinsic parameters from several views of a calibration pattern.

```
CalibrateCamera2(objectPoints,imagePoints,pointCounts,imageSize,cameraMatrix,distCo
None
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints The joint matrix of object points projections in the camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints` (=N).

imageSize Size of the image, used only to initialize the intrinsic camera matrix

cameraMatrix The output 3x3 floating-point camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$.
If `CV_CALIB_USE_INTRINSIC_GUESS` and/or `CV_CALIB_FIX_ASPECT_RATIO` are specified, some or all of `fx, fy, cx, cy` must be initialized before calling the function

distCoeffs The output 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

rvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of rotation vectors (see cv.Rodrigues2), estimated for each pattern view. That is, each k-th rotation vector together with the corresponding k-th translation vector (see the next output parameter description) brings the calibration pattern from the model coordinate space (in which object points are specified) to the world coordinate space, i.e. real position of the calibration pattern in the k-th pattern view (k=0..*M*-1)

tvecs The output 3x*M* or *M*x3 1-channel, or 1x*M* or *M*x1 3-channel array of translation vectors, estimated for each pattern view.

flags Different flags, may be 0 or combination of the following values:

- **CV_CALIB_USE_INTRINSIC_GUESS** `cameraMatrix` contains the valid initial values of `fx, fy, cx, cy` that are optimized further. Otherwise, `(cx, cy)` is initially set to the image center (`imageSize` is used here), and focal distances are computed in some least-squares fashion. Note, that if intrinsic parameters are known, there is no need to use this function just to estimate the extrinsic parameters. Use cv.FindExtrinsicCameraParams2 instead.
- **CV_CALIB_FIX_PRINCIPAL_POINT** The principal point is not changed during the global optimization, it stays at the center or at the other location specified when `CV_CALIB_USE_INTRINSIC_GUESS` is set too.
- **CV_CALIB_FIX_ASPECT_RATIO** The functions considers only `fy` as a free parameter, the ratio `fx/fy` stays the same as in the input `cameraMatrix`.
  When `CV_CALIB_USE_INTRINSIC_GUESS` is not set, the actual input values of `fx` and `fy` are ignored, only their ratio is computed and used further.
- **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients (p_1, p_2) will be set to zeros and stay zero.

The function estimates the intrinsic camera parameters and extrinsic parameters for each of the views. The coordinates of 3D object points and their correspondent 2D projections in each view must be specified. That may be achieved by using an object with known geometry and easily detectable feature points. Such an object is called a calibration rig or calibration pattern, and OpenCV has built-in support for a chessboard as a calibration rig (see cv.FindChessboardCorners). Currently, initialization of intrinsic parameters (when `CV_CALIB_USE_INTRINSIC_GUESS` is not set) is only implemented for planar calibration patterns (where z-coordinates of the object points must be all 0's). 3D calibration rigs can also be used as long as initial `cameraMatrix` is provided.

The algorithm does the following:

1. First, it computes the initial intrinsic parameters (the option only available for planar calibration patterns) or reads them from the input parameters. The distortion coefficients are all set to zeros initially (unless some of `CV_CALIB_FIX_K?` are specified).

2. The the initial camera pose is estimated as if the intrinsic parameters have been already known. This is done using cv.FindExtrinsicCameraParams2

3. After that the global Levenberg-Marquardt optimization algorithm is run to minimize the reprojection error, i.e. the total sum of squared distances between the observed feature points `imagePoints` and the projected (using the current estimates for camera parameters and the poses) object points `objectPoints`; see cv.ProjectPoints2.

Note: if you're using a non-square (=non-NxN) grid and cv. for calibration, and `calibrateCamera` returns bad values (i.e. zero distortion coefficients, an image center very far from $(w/2-0.5, h/2-0.5)$, and / or large differences between f_x and f_y (ratios of 10:1 or more)), then you've probaby used `patternSize=cvSize(rows,cols)`, but should use `patternSize=cvSize(cols,rows)` in cv.FindChessboardCorners.

See also: cv.FindChessboardCorners, cv.FindExtrinsicCameraParams2, cv., cv.StereoCalibrate, cv.Undistort2

ComputeCorrespondEpilines

computeCorrespondEpilines For points in one image of a stereo pair, computes the corresponding epilines in the other image.

```
ComputeCorrespondEpilines(points, whichImage, F, lines) -> None
```

points The input points. `2xN, Nx2, 3xN` or `Nx3` array (where `N` number of points). Multi-channel `1xN` or `Nx1` array is also acceptable

whichImage Index of the image (1 or 2) that contains the `points`

F The fundamental matrix that can be estimated using cv.FindFundamentalMat or cv.StereoRectify.

lines The output epilines, a `3xN` or `Nx3` array. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c)

For every point in one of the two images of a stereo-pair the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see cv.FindFundamentalMat), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (i.e. when `whichImage=1`) is computed as:

$$l_i^{(2)} = F p_i^{(1)}$$

and, vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized, such that $a_i^2 + b_i^2 = 1$.

ConvertPointsHomogeneous

convertPointsHomogeneous Convert points to/from homogeneous coordinates.

```
ConvertPointsHomogeneous( src, dst ) -> None
```

src The input array or vector of 2D or 3D points

dst The output vector of 3D or 2D points, respectively

The 2D or 3D points from/to homogeneous coordinates, or simply the array. If the input array dimensionality is larger than the output, each coordinate is divided by the last coordinate:

$$(x, y[, z], w) -> (x', y'[, z'])$$
where
$x' = x/w$
$y' = y/w$
$z' = z/w$ (if output is 3D)

If the output array dimensionality is larger, an extra 1 is appended to each point. Otherwise, the input array is simply copied (with optional transposition) to the output.

cv.CreatePOSITObject

Initializes a structure containing object information.

```
CreatePOSITObject(points)-> POSITObject
```

points List of 3D points

The function allocates memory for the object structure and computes the object inverse matrix.

The preprocessed object data is stored in the structure cv.CvPOSITObject, internal for OpenCV, which means that the user cannot directly access the structure data. The user may only create this structure and pass its pointer to the function.

An object is defined as a set of points given in a coordinate system. The function cv.POSIT computes a vector that begins at a camera-related coordinate system center and ends at the `points[0]` of the object.

Once the work with a given object is finished, the function cv.ReleasePOSITObject must be called to free memory.

cv.CreateStereoBMState

Creates block matching stereo correspondence structure.

```
CreateStereoBMState(preset=CV_STEREO_BM_BASIC,numberOfDisparities=0)->
StereoBMState
```

preset ID of one of the pre-defined parameter sets. Any of the parameters can be overridden after creating the structure.

numberOfDisparities The number of disparities. If the parameter is 0, it is taken from the preset, otherwise the supplied value overrides the one from preset.

```
#define CV_STEREO_BM_BASIC 0
#define CV_STEREO_BM_FISH_EYE 1
#define CV_STEREO_BM_NARROW 2
```

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cv.FindStereoCorrespondenceBM.

cv.CreateStereoGCState

Creates the state of graph cut-based stereo correspondence algorithm.

```
CreateStereoGCState(numberOfDisparities,maxIters)-> StereoGCState
```

numberOfDisparities The number of disparities. The disparity search range will be state->minDisparity
$disparity <$ state->minDisparity + state->numberOfDisparities

maxIters Maximum number of iterations. On each iteration all possible (or reasonable) alpha-expansions are tried. The algorithm may terminate earlier if it could not find an alpha-expansion that decreases the overall cost function value. See [?] for details.

The function creates the stereo correspondence structure and initializes it. It is possible to override any of the parameters at any time between the calls to cv.FindStereoCorrespondenceGC.

CvStereoBMState

The structure for block matching stereo correspondence algorithm.

```
typedef struct CvStereoBMState
{
    //pre filters (normalize input images):
    int       preFilterType; // 0 for now
    int       preFilterSize; // ~5x5..21x21
    int       preFilterCap;  // up to ~31
    //correspondence using Sum of Absolute Difference (SAD):
    int       SADWindowSize; // Could be 5x5..21x21
    int       minDisparity;  // minimum disparity (=0)
    int       numberOfDisparities; // maximum disparity - minimum disparity
    //post filters (knock out bad matches):
    int       textureThreshold; // areas with no texture are ignored
    float     uniquenessRatio;// filter out pixels if there are other close matches
                              // with different disparity
    int       speckleWindowSize; // the maximum area of speckles to remove
                                 // (set to 0 to disable speckle filtering)
    int       speckleRange; // acceptable range of disparity variation in each connected co
    // internal data
    ...
}
CvStereoBMState;
```

The block matching stereo correspondence algorithm, by Kurt Konolige, is very fast one-pass stereo matching algorithm that uses sliding sums of absolute differences between pixels in the left image and the pixels in the right image, shifted by some varying amount of pixels (from `minDisparity` to `minDisparity+numberOfDisparities`). On a pair of images WxH the algorithm computes disparity in `O(W*H*numberOfDisparities)` time. In order to improve quality and reability of the disparity map, the algorithm includes pre-filtering and post-filtering procedures.

Note that the algorithm searches for the corresponding blocks in x direction only. It means that the supplied stereo pair should be rectified. Vertical stereo layout is not directly supported, but in such a case the images could be transposed by user.

CvStereoGCState

The structure for graph cuts-based stereo correspondence algorithm

```
typedef struct CvStereoGCState
{
    int Ithreshold; // threshold for piece-wise linear data cost function (5 by default)
    int interactionRadius; // radius for smoothness cost function (1 by default; means Pott
    float K, lambda, lambda1, lambda2; // parameters for the cost function
                                       // (usually computed adaptively from the input data)
    int occlusionCost; // 10000 by default
    int minDisparity; // 0 by default; see CvStereoBMState
    int numberOfDisparities; // defined by user; see CvStereoBMState
    int maxIters; // number of iterations; defined by user.

    // internal buffers
    CvMat* left;
    CvMat* right;
    CvMat* dispLeft;
    CvMat* dispRight;
    CvMat* ptrLeft;
    CvMat* ptrRight;
    CvMat* vtxBuf;
    CvMat* edgeBuf;
}
CvStereoGCState;
```

The graph cuts stereo correspondence algorithm, described in [?] (as **KZ1**), is non-realtime stereo correpsondence algorithm that usually gives very accurate depth map with well-defined object boundaries. The algorithm represents stereo problem as a sequence of binary optimization problems, each of those is solved using maximum graph flow algorithm. The state structure above should not be allocated and initialized manually; instead, use cv.CreateStereoGCState and then override necessary parameters if needed.

DecomposeProjectionMatrix

decomposeProjectionMatrix Decomposes the projection matrix into a rotation matrix and a camera matrix.

```
DecomposeProjectionMatrix(projMatrix, cameraMatrix, rotMatrix,
transVect, rotMatrX = None, rotMatrY = None, rotMatrZ = None) ->
eulerAngles
```

projMatrix The 3x4 input projection matrix P

cameraMatrix The output 3x3 camera matrix K

rotMatrix The output 3x3 external rotation matrix R

transVect The output 4x1 translation vector T

rotMatrX Optional 3x3 rotation matrix around x-axis

rotMatrY Optional 3x3 rotation matrix around y-axis

rotMatrZ Optional 3x3 rotation matrix around z-axis

eulerAngles Optional 3 points containing the three Euler angles of rotation

The function computes a decomposition of a projection matrix into a calibration and a rotation matrix and the position of the camera.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

The function is based on cv.RQDecomp3x3.

DrawChessboardCorners

drawChessboardCorners Renders the detected chessboard corners.

```
DrawChessboardCorners(image,patternSize,corners,patternWasFound)-> None
```

image The destination image; it must be an 8-bit color image

patternSize The number of inner corners per chessboard row and column. (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The array of corners detected

patternWasFound Indicates whether the complete board was found $(\neq 0)$ or not $(= 0)$. One may just pass the return value cv.FindChessboardCornersfindChessboardCorners here

The function draws the individual chessboard corners detected as red circles if the board was not found or as colored corners connected with lines if the board was found.

FindChessboardCorners

findChessboardCorners Finds the positions of the internal corners of the chessboard.

```
FindChessboardCorners(image, patternSize, flags=CV_CALIB_CB_ADAPTIVE_THRESH)
-> corners
```

image Source chessboard view; it must be an 8-bit grayscale or color image

patternSize The number of inner corners per chessboard row and column (patternSize = cvSize(points_per_row,points_per_colum) = cvSize(columns,rows))

corners The output array of corners detected

flags Various operation flags, can be 0 or a combination of the following values:

- **CV_CALIB_CB_ADAPTIVE_THRESH** use adaptive thresholding to convert the image to black and white, rather than a fixed threshold level (computed from the average image brightness).
- **CV_CALIB_CB_NORMALIZE_IMAGE** normalize the image gamma with cv.EqualizeHist before applying fixed or adaptive thresholding.
- **CV_CALIB_CB_FILTER_QUADS** use additional criteria (like contour area, perimeter, square-like shape) to filter out false quads that are extracted at the contour retrieval stage.

The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners have been found and they have been placed in a certain order (row by row, left to right in every row), otherwise, if the function fails to find all the corners or reorder them, it returns 0. For example, a regular chessboard has 8 x 8 squares and 7 x 7 internal corners, that is, points, where

the black squares touch each other. The coordinates detected are approximate, and to determine their position more accurately, the user may use the function cv.FindCornerSubPix.

Note: the function requires some white space (like a square-thick border, the wider the better) around the board to make the detection more robust in various environment (otherwise if there is no border and the background is dark, the outer black squares could not be segmented properly and so the square grouping and ordering algorithm will fail).

FindExtrinsicCameraParams2

solvePnP Finds the object pose from the 3D-2D point correspondences

```
FindExtrinsicCameraParams2(objectPoints,imagePoints,cameraMatrix,distCoeffs,rvec,tv
None
```

objectPoints The array of object points in the object coordinate space, 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel, where N is the number of points.

imagePoints The array of corresponding image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel, where N is the number of points.

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$. If it is NULL, all of the distortion coefficients are set to 0

rvec The output rotation vector (see cv.Rodrigues2) that (together with `tvec`) brings points from the model coordinate system to the camera coordinate system

tvec The output translation vector

The function estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. This function finds such a pose that minimizes reprojection error, i.e. the sum of squared distances between the observed projections `imagePoints` and the projected (using cv.ProjectPoints2) `objectPoints`.

FindFundamentalMat

findFundamentalMat Calculates the fundamental matrix from the corresponding points in two images.

```
FindFundamentalMat(points1, points2, fundamentalMatrix,
method=CV_FM_RANSAC, param1=1., param2=0.99, status = None) -> None
```

points1 Array of `N` points from the first image. It can be `2xN, Nx2, 3xN` or `Nx3` 1-channel array or `1xN` or `Nx1` 2- or 3-channel array . The point coordinates should be floating-point (single or double precision)

points2 Array of the second image points of the same size and format as `points1`

fundamentalMatrix The output fundamental matrix or matrices. The size should be 3x3 or 9x3 (7-point method may return up to 3 matrices)

method Method for computing the fundamental matrix

CV_FM_7POINT for a 7-point algorithm. $N = 7$

CV_FM_8POINT for an 8-point algorithm. $N \geq 8$

CV_FM_RANSAC for the RANSAC algorithm. $N \geq 8$

CV_FM_LMEDS for the LMedS algorithm. $N \geq 8$

param1 The parameter is used for RANSAC. It is the maximum distance from point to epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution and the image noise

param2 The parameter is used for RANSAC or LMedS methods only. It specifies the desirable level of confidence (probability) that the estimated matrix is correct

status The optional output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in RANSAC and LMedS methods. For other methods it is set to all 1's

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the number of fundamental matrices found (1 or 3) and 0, if no matrix is found . Normally just 1 matrix is found, but in the case of 7-point algorithm the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

The calculated fundamental matrix may be passed further to cv.ComputeCorrespondEpilines that finds the epipolar lines corresponding to the specified points. It can also be passed to cv.StereoRectifyUncalibrated to compute the rectification transformation.

FindHomography

findHomography Finds the perspective transformation between two planes.

```
FindHomography(srcPoints,dstPoints,H,method,ransacReprojThreshold=0.0,
status=None)-> H
```

srcPoints Coordinates of the points in the original plane, 2xN, Nx2, 3xN or Nx3 1-channel array (the latter two are for representation in homogeneous coordinates), where N is the number of points. 1xN or Nx1 2- or 3-channel array can also be passed.

dstPoints Point coordinates in the destination plane, 2xN, Nx2, 3xN or Nx3 1-channel, or 1xN or Nx1 2- or 3-channel array.

H The output 3x3 homography matrix

method The method used to computed homography matrix; one of the following:

- **0** a regular method using all the points
- **CV_RANSAC** RANSAC-based robust method
- **CV_LMEDS** Least-Median robust method

ransacReprojThreshold The maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointHomogeneous}(\texttt{HsrcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range 1 to 10.

status The optional output mask set by a robust method (CV_RANSAC or CV_LMEDS). *Note that the input mask values are ignored.*

The function finds the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

So that the back-projection error

$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter method is set to the default value 0, the function uses all the point pairs to compute the initial homography estimate with a simple least-squares scheme.

However, if not all of the point pairs ($srcPoints_i$, $dstPoints_i$) fit the rigid perspective transformation (i.e. there are some outliers), this initial estimate will be poor. In this case one can use one of the 2 robust methods. Both methods, RANSAC and LMeDS, try many different random subsets of the corresponding point pairs (of 4 pairs each), estimate the homography matrix using this subset and a simple least-square algorithm and then compute the quality/goodness of the computed homography (which is the number of inliers for RANSAC or the median re-projection error for LMeDs). The best subset is then used to produce the initial estimate of the homography matrix and the mask of inliers/outliers.

Regardless of the method, robust or not, the computed homography matrix is refined further (using inliers only in the case of a robust method) with the Levenberg-Marquardt method in order to reduce the re-projection error even more.

The method RANSAC can handle practically any ratio of outliers, but it needs the threshold to distinguish inliers from outliers. The method LMeDS does not need any threshold, but it works correctly only when there are more than 50% of inliers. Finally, if you are sure in the computed features, where can be only some small noise present, but no outliers, the default method could be the best choice.

The function is used to find initial intrinsic and extrinsic matrices. Homography matrix is determined up to a scale, thus it is normalized so that $h_{33} = 1$.

See also: cv.GetAffineTransform, cv.GetPerspectiveTransform, cv.EstimateRigidMotion, cv.WarpPerspective, cv.PerspectiveTransform

---

cv.FindStereoCorrespondenceBM

Computes the disparity map using block matching algorithm.

```
FindStereoCorrespondenceBM(left,right,disparity,state)-> None
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

disparity The output single-channel 16-bit signed, or 32-bit floating-point disparity map of the same size as input images. In the first case the computed disparities are represented as fixed-point numbers with 4 fractional bits (i.e. the computed disparity values are multiplied by 16 and rounded to integers).

state Stereo correspondence structure.

The function cvFindStereoCorrespondenceBM computes disparity map for the input rectified stereo pair. Invalid pixels (for which disparity can not be computed) are set to `state->minDisparity - 1` (or to `(state->minDisparity-1)*16` in the case of 16-bit fixed-point disparity map)

---

cv.FindStereoCorrespondenceGC

Computes the disparity map using graph cut-based algorithm.

```
FindStereoCorrespondenceGC( left, right, dispLeft, dispRight, state,
useDisparityGuess=(0))-> None
```

left The left single-channel, 8-bit image.

right The right image of the same size and the same type.

dispLeft The optional output single-channel 16-bit signed left disparity map of the same size as input images.

dispRight The optional output single-channel 16-bit signed right disparity map of the same size as input images.

state Stereo correspondence structure.

useDisparityGuess If the parameter is not zero, the algorithm will start with pre-defined disparity maps. Both dispLeft and dispRight should be valid disparity maps. Otherwise, the function starts with blank disparity maps (all pixels are marked as occlusions).

The function computes disparity maps for the input rectified stereo pair. Note that the left disparity image will contain values in the following range:

$$-\texttt{state->numberOfDisparities}-\texttt{state->minDisparity} < dispLeft(x,y) \leq -\texttt{state->minDisparity}$$

or

$$dispLeft(x,y) == \texttt{CV\_STEREO\_GC\_OCCLUSION}$$

and for the right disparity image the following will be true:

$$\texttt{state->minDisparity} \leq dispRight(x,y) < \texttt{state->minDisparity}+\texttt{state->numberOfDisparities}$$

or

$$dispRight(x,y) == \texttt{CV\_STEREO\_GC\_OCCLUSION}$$

that is, the range for the left disparity image will be inversed, and the pixels for which no good match has been found, will be marked as occlusions.

Here is how the function can be called:

```
// image_left and image_right are the input 8-bit single-channel images
// from the left and the right cameras, respectively
CvSize size = cvGetSize(image_left);
CvMat* disparity_left = cvCreateMat( size.height, size.width, CV_16S );
CvMat* disparity_right = cvCreateMat( size.height, size.width, CV_16S );
CvStereoGCState* state = cvCreateStereoGCState( 16, 2 );
cvFindStereoCorrespondenceGC( image_left, image_right,
    disparity_left, disparity_right, state, 0 );
cvReleaseStereoGCState( &state );
// now process the computed disparity images as you want ...
```

and this is the output left disparity image computed from the well-known Tsukuba stereo pair and multiplied by -16 (because the values in the left disparity images are usually negative):

```
CvMat* disparity_left_visual = cvCreateMat( size.height, size.width, CV_8U );
cvConvertScale( disparity_left, disparity_left_visual, -16 );
cvSave( "disparity.pgm", disparity_left_visual );
```

GetOptimalNewCameraMatrix

getOptimalNewCameraMatrix Returns the new camera matrix based on the free scaling parameter

```
GetOptimalNewCameraMatrix(cameraMatrix, distCoeffs, imageSize, alpha,
newCameraMatrix, newImageSize=(0,0), validPixROI=0) -> None
```

cameraMatrix The input camera matrix

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

imageSize The original image size

alpha The free scaling parameter between 0 (when all the pixels in the undistorted image will be valid) and 1 (when all the source image pixels will be retained in the undistorted image); see cv.StereoRectify

newCameraMatrix The output new camera matrix.

newImageSize The image size after rectification. By default it will be set to `imageSize`.

validPixROI The optional output rectangle that will outline all-good-pixels region in the undistorted image. See `roi1, roi2` description in cv.StereoRectify

The function computes the optimal new camera matrix based on the free scaling parameter. By varying this parameter the user may retrieve only sensible pixels `alpha=0`, keep all the original image pixels if there is valuable information in the corners `alpha=1`, or get something in between. When `alpha>0`, the undistortion result will likely have some black pixels corresponding to "virtual" pixels outside of the captured distorted image. The original camera matrix, distortion coefficients, the computed new camera matrix and the `newImageSize` should be passed to cv.InitUndistortRectifyMap to produce the maps for cv.Remap.

InitIntrinsicParams2D

initCameraMatrix2D Finds the initial camera matrix from the 3D-2D point correspondences

```
InitIntrinsicParams2D(objectPoints, imagePoints, npoints, imageSize,
cameraMatrix, aspectRatio=1.) -> None
```

objectPoints The joint array of object points; see cv.CalibrateCamera2

imagePoints The joint array of object point projections; see cv.CalibrateCamera2

npoints The array of point counts; see cv.CalibrateCamera2

imageSize The image size in pixels; used to initialize the principal point

cameraMatrix The output camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

aspectRatio If it is zero or negative, both f_x and f_y are estimated independently. Otherwise $f_x = f_y * \texttt{aspectRatio}$

The function estimates and returns the initial camera matrix for camera calibration process. Currently, the function only supports planar calibration patterns, i.e. patterns where each object point has z-coordinate =0.

InitUndistortMap

Computes an undistortion map.

```
InitUndistortMap(cameraMatrix,distCoeffs,map1,map2)-> None
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function is a simplified variant of cv.InitUndistortRectifyMap where the rectification transformation `R` is identity matrix and `newCameraMatrix=cameraMatrix`.

InitUndistortRectifyMap

initUndistortRectifyMap Computes the undistortion and rectification transformation map.

```
InitUndistortRectifyMap(cameraMatrix,distCoeffs,R,newCameraMatrix,map1,map2)->
None
```

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$.

R The optional rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv.StereoRectify can be passed here. If the matrix is NULL , the identity transformation is assumed

newCameraMatrix The new camera matrix $A' = \begin{bmatrix} f'_x & 0 & c'_x \\ 0 & f'_y & c'_y \\ 0 & 0 & 1 \end{bmatrix}$

map1 The first output map of type `CV_32FC1` or `CV_16SC2` - the second variant is more efficient

map2 The second output map of type `CV_32FC1` or `CV_16UC1` - the second variant is more efficient

The function computes the joint undistortion+rectification transformation and represents the result in the form of maps for cv.Remap. The undistorted image will look like the original, as if it was captured with a camera with camera matrix `=newCameraMatrix` and zero distortion. In the case of monocular camera `newCameraMatrix` is usually equal to `cameraMatrix`, or it can be computed by cv.GetOptimalNewCameraMatrix for a better control over scaling. In the case of stereo camera `newCameraMatrix` is normally set to `P1` or `P2` computed by cv.StereoRectify.

Also, this new camera will be oriented differently in the coordinate space, according to `R`. That, for example, helps to align two heads of a stereo camera so that the epipolar lines on both images become horizontal and have the same y- coordinate (in the case of horizontally aligned stereo camera).

The function actually builds the maps for the inverse mapping algorithm that is used by cv.Remap. That is, for each pixel (u, v) in the destination (corrected and rectified) image the function computes the corresponding coordinates in the source image (i.e. in the original image from camera). The process is the following:

$$
\begin{aligned}
&x \leftarrow (u - {c'}_x)/{f'}_x \\
&y \leftarrow (v - {c'}_y)/{f'}_y \\
&[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T \\
&x' \leftarrow X/W \\
&y' \leftarrow Y/W \\
&x" \leftarrow x'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x' y' + p_2(r^2 + 2x'^2) \\
&y" \leftarrow y'(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y'^2) + 2p_2 x' y' \\
&map_x(u, v) \leftarrow x" f_x + c_x \\
&map_y(u, v) \leftarrow y" f_y + c_y
\end{aligned}
$$

where $(k_1, k_2, p_1, p_2[, k_3])$ are the distortion coefficients.

In the case of a stereo camera this function is called twice, once for each camera head, after cv.StereoRectify, which in its turn is called after cv.StereoCalibrate. But if the stereo camera was not calibrated, it is still possible to compute the rectification transformations directly from the fundamental matrix using cv.StereoRectifyUncalibrated. For each camera the function computes homography `H` as the rectification transformation in pixel domain, not a rotation matrix `R` in 3D space. The `R` can be computed from `H` as

$$\texttt{R} = \texttt{cameraMatrix}^{-1} \cdot \texttt{H} \cdot \texttt{cameraMatrix}$$

where the `cameraMatrix` can be chosen arbitrarily.

cv.POSIT

Implements the POSIT algorithm.

```
POSIT(posit_object,imagePoints,focal_length,criteria)->
(rotationMatrix,translation_vector)
```

posit_object Pointer to the object structure

imagePoints Pointer to the object points projections on the 2D image plane

focal_length Focal length of the camera used

criteria Termination criteria of the iterative POSIT algorithm

rotationMatrix Matrix of rotations

translation_vector Translation vector

The function implements the POSIT algorithm. Image coordinates are given in a camera-related coordinate system. The focal length may be retrieved using the camera calibration functions. At every iteration of the algorithm a new perspective projection of the estimated pose is computed.

Difference norm between two projections is the maximal distance between corresponding points. The parameter `criteria.epsilon` serves to stop the algorithm if the difference is small.

ProjectPoints2

projectPoints Project 3D points on to an image plane.

```
ProjectPoints2(objectPoints,rvec,tvec,cameraMatrix,distCoeffs,
imagePoints,dpdrot=NULL,dpdt=NULL,dpdf=NULL,dpdc=NULL,dpddist=NULL)->
None
```

objectPoints The array of object points, 3xN or Nx3 1-channel or 1xN or Nx1 3-channel , where N is the number of points in the view

rvec The rotation vector, see cv.Rodrigues2

tvec The translation vector

cameraMatrix The camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The input 4x1, 1x4, 5x1 or 1x5 vector of distortion coefficients $(k_1, k_2, p_1, p_2[, k_3])$. If it is None, all of the distortion coefficients are considered 0's

imagePoints The output array of image points, 2xN or Nx2 1-channel or 1xN or Nx1 2-channel

dpdrot Optional 2Nx3 matrix of derivatives of image points with respect to components of the rotation vector

dpdt Optional 2Nx3 matrix of derivatives of image points with respect to components of the translation vector

dpdf Optional 2Nx2 matrix of derivatives of image points with respect to f_x and f_y

dpdc Optional 2Nx2 matrix of derivatives of image points with respect to c_x and c_y

dpddist Optional 2Nx4 matrix of derivatives of image points with respect to distortion coefficients

The function computes projections of 3D points to the image plane given intrinsic and extrinsic camera parameters. Optionally, the function computes jacobians - matrices of partial derivatives of image points coordinates (as functions of all the input parameters) with respect to the particular parameters, intrinsic and/or extrinsic. The jacobians are used during the global optimization in cv.CalibrateCamera2, cv.FindExtrinsicCameraParams2 and cv.StereoCalibrate. The function itself can also used to compute re-projection error given the current intrinsic and extrinsic parameters.

Note, that by setting `rvec=tvec=(0,0,0)`, or by setting `cameraMatrix` to 3x3 identity matrix, or by passing zero distortion coefficients, you can get various useful partial cases of the function, i.e. you can compute the distorted coordinates for a sparse set of points, or apply a perspective transformation (and also compute the derivatives) in the ideal zero-distortion setup etc.

ReprojectImageTo3D

reprojectImageTo3D Reprojects disparity image to 3D space.

```
ReprojectImageTo3D(disparity, _3dImage, Q, handleMissingValues=0) ->
None
```

disparity The input single-channel 16-bit signed or 32-bit floating-point disparity image

_3dImage The output 3-channel floating-point image of the same size as `disparity`. Each element of `_3dImage(x,y)` will contain the 3D coordinates of the point `(x,y)`, computed from the disparity map.

Q The 4×4 perspective transformation matrix that can be obtained with cv.StereoRectify

handleMissingValues If true, when the pixels with the minimal disparity (that corresponds to the ouliers; see cv.FindStereoCorrespondenceBM) will be transformed to 3D points with some very large Z value (currently set to 10000)

The function transforms 1-channel disparity map to 3-channel image representing a 3D surface. That is, for each pixel `(x,y)` and the corresponding disparity `d=disparity(x,y)` it computes:

$$[X\ Y\ Z\ W]^T = \mathtt{Q} * [x\ y\ \mathtt{disparity}(x,y)\ 1]^T$$
$$\mathtt{\_3dImage}(x,y) = (X/W,\ Y/W,\ Z/W)$$

The matrix `Q` can be arbitrary 4×4 matrix, e.g. the one computed by cv.StereoRectify. To reproject a sparse set of points (x,y,d),... to 3D space, use cv.PerspectiveTransform.

RQDecomp3x3

RQDecomp3x3 Computes the 'RQ' decomposition of 3x3 matrices.

```
RQDecomp3x3(M, R, Q, Qx = None, Qy = None, Qz = None) -> eulerAngles
```

M The 3x3 input matrix

R The output 3x3 upper-triangular matrix

Q The output 3x3 orthogonal matrix

Qx Optional 3x3 rotation matrix around x-axis

Qy Optional 3x3 rotation matrix around y-axis

Qz Optional 3x3 rotation matrix around z-axis

eulerAngles Optional three Euler angles of rotation

The function computes a RQ decomposition using the given rotations. This function is used in cv.DecomposeProjectionMatrix to decompose the left 3x3 submatrix of a projection matrix into a camera and a rotation matrix.

It optionally returns three rotation matrices, one for each axis, and the three Euler angles that could be used in OpenGL.

Rodrigues2

Rodrigues Converts a rotation matrix to a rotation vector or vice versa.

```
Rodrigues2(src,dst,jacobian=0)-> None
```

src The input rotation vector (3x1 or 1x3) or rotation matrix (3x3)

dst The output rotation matrix (3x3) or rotation vector (3x1 or 1x3), respectively

jacobian Optional output Jacobian matrix, 3x9 or 9x3 - partial derivatives of the output array components with respect to the input array components

$$\theta \leftarrow norm(r)$$
$$r \leftarrow r/\theta$$
$$R = \cos\theta I + (1 - \cos\theta) r r^T + \sin\theta \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

Inverse transformation can also be done easily, since

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

A rotation vector is a convenient and most-compact representation of a rotation matrix (since any rotation matrix has just 3 degrees of freedom). The representation is used in the global 3D geometry optimization procedures like cv.CalibrateCamera2, cv.StereoCalibrate or cv.FindExtrinsicCameraParams

---

StereoCalibrate

stereoCalibrate Calibrates stereo camera.

```
StereoCalibrate( objectPoints, imagePoints1, imagePoints2, pointCounts,
cameraMatrix1, distCoeffs1, cameraMatrix2, distCoeffs2, imageSize, R,
T, E=NULL, F=NULL, term_crit=(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,30,1e-6),
flags=CV_CALIB_FIX_INTRINSIC)-> None
```

objectPoints The joint matrix of object points - calibration pattern features in the model coordinate space. It is floating-point 3xN or Nx3 1-channel, or 1xN or Nx1 3-channel array, where N is the total number of points in all views.

imagePoints1 The joint matrix of object points projections in the first camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

imagePoints2 The joint matrix of object points projections in the second camera views. It is floating-point 2xN or Nx2 1-channel, or 1xN or Nx1 2-channel array, where N is the total number of points in all views

pointCounts Integer 1xM or Mx1 vector (where M is the number of calibration pattern views) containing the number of points in each particular view. The sum of vector elements must match the size of `objectPoints` and `imagePoints*` (=N).

cameraMatrix1 The input/output first camera matrix: $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$, $j = 0, 1$. If any of `CV_CALIB_USE_INTRINSIC_GUESS`, `CV_CALIB_FIX_ASPECT_RATIO`, `CV_CALIB_FIX_INTRINSIC` or `CV_CALIB_FIX_FOCAL_LENGTH` are specified, some or all of the matrices' components must be initialized; see the flags description

distCoeffs1 The input/output lens distortion coefficients for the first camera, 4x1, 5x1, 1x4 or 1x5 floating-point vectors $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$, $j = 0, 1$. If any of `CV_CALIB_FIX_K1`, `CV_CALIB_FIX_K2` or `CV_CALIB_FIX_K3` is specified, then the corresponding elements of the distortion coefficients must be initialized.

cameraMatrix2 The input/output second camera matrix, as cameraMatrix1.

distCoeffs2 The input/output lens distortion coefficients for the second camera, as distCoeffs1.

imageSize Size of the image, used only to initialize intrinsic camera matrix.

R The output rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The output translation vector between the cameras' coordinate systems.

E The optional output essential matrix.

F The optional output fundamental matrix.

term_crit The termination criteria for the iterative optimization algorithm.

flags Different flags, may be 0 or combination of the following values:

- **CV_CALIB_FIX_INTRINSIC** If it is set, `cameraMatrix?`, as well as `distCoeffs?` are fixed, so that only `R, T, E` and `F` are estimated.
- **CV_CALIB_USE_INTRINSIC_GUESS** The flag allows the function to optimize some or all of the intrinsic parameters, depending on the other flags, but the initial values are provided by the user.
- **CV_CALIB_FIX_PRINCIPAL_POINT** The principal points are fixed during the optimization.
- **CV_CALIB_FIX_FOCAL_LENGTH** $f_x^{(j)}$ and $f_y^{(j)}$ are fixed.
- **CV_CALIB_FIX_ASPECT_RATIO** $f_y^{(j)}$ is optimized, but the ratio $f_x^{(j)}/f_y^{(j)}$ is fixed.
- **CV_CALIB_SAME_FOCAL_LENGTH** Enforces $f_x^{(0)} = f_x^{(1)}$ and $f_y^{(0)} = f_y^{(1)}$
- **CV_CALIB_ZERO_TANGENT_DIST** Tangential distortion coefficients for each camera are set to zeros and fixed there.
- **CV_CALIB_FIX_K1, CV_CALIB_FIX_K2, CV_CALIB_FIX_K3** Fixes the corresponding radial distortion coefficient (the coefficient must be passed to the function)

The function estimates transformation between the 2 cameras making a stereo pair. If we have a stereo camera, where the relative position and orientatation of the 2 cameras is fixed, and if we computed poses of an object relative to the fist camera and to the second camera, (R1, T1) and (R2, T2), respectively (that can be done with cv.FindExtrinsicCameraParams2), obviously, those poses will relate to each other, i.e. given (R_1, T_1) it should be possible to compute (R_2, T_2) - we only need to know the position and orientation of the 2nd camera relative to the 1st camera. That's what the described function does. It computes (R, T) such that:

$$R_2 = R * R_1 T_2 = R * T_1 + T,$$

Optionally, it computes the essential matrix E:

$$E = \begin{bmatrix} 0 & -T_2 & T_1 \\ T_2 & 0 & -T_0 \\ -T_1 & T_0 & 0 \end{bmatrix} * R$$

where T_i are components of the translation vector T: $T = [T_0, T_1, T_2]^T$. And also the function can compute the fundamental matrix F:

$$F = cameraMatrix2^{-T} E cameraMatrix1^{-1}$$

Besides the stereo-related information, the function can also perform full calibration of each of the 2 cameras. However, because of the high dimensionality of the parameter space and noise in the input data the function can diverge from the correct solution. Thus, if intrinsic parameters can be estimated with high accuracy for each of the cameras individually (e.g. using cv.CalibrateCamera2), it is recommended to do so and then pass `CV_CALIB_FIX_INTRINSIC` flag to the function along with the computed intrinsic parameters. Otherwise, if all the parameters are estimated at once, it makes sense to restrict some parameters, e.g. pass `CV_CALIB_SAME_FOCAL_LENGTH` and `CV_CALIB_ZERO_TANGENT_DIST` flags, which are usually reasonable assumptions.

Similarly to cv.CalibrateCamera2, the function minimizes the total re-projection error for all the points in all the available views from both cameras.

StereoRectify

stereoRectify Computes rectification transforms for each head of a calibrated stereo camera.

```
StereoRectify( cameraMatrix1, cameraMatrix2, distCoeffs1, distCoeffs2,
imageSize, R, T, R1, R2, P1, P2, Q=NULL, flags=CV_CALIB_ZERO_DISPARITY,
alpha=-1, newImageSize=(0,0))-> (roi1, roi2)
```

cameraMatrix1, cameraMatrix2 The camera matrices $\begin{bmatrix} f_x^{(j)} & 0 & c_x^{(j)} \\ 0 & f_y^{(j)} & c_y^{(j)} \\ 0 & 0 & 1 \end{bmatrix}$.

distCoeffs1, distCoeffs2 The input distortion coefficients for each camera, ${k_1}^{(j)}, {k_2}^{(j)}, {p_1}^{(j)}, {p_2}^{(j)}[, {k_3}^{(j)}]$

imageSize Size of the image used for stereo calibration.

R The rotation matrix between the 1st and the 2nd cameras' coordinate systems.

T The translation vector between the cameras' coordinate systems.

R1, R2 The output 3×3 rectification transforms (rotation matrices) for the first and the second cameras, respectively.

P1, P2 The output 3×4 projection matrices in the new (rectified) coordinate systems.

Q The output 4×4 disparity-to-depth mapping matrix, see cv..

flags The operation flags; may be 0 or `CV_CALIB_ZERO_DISPARITY`. If the flag is set, the function makes the principal points of each camera have the same pixel coordinates in the rectified views. And if the flag is not set, the function may still shift the images in horizontal or vertical direction (depending on the orientation of epipolar lines) in order to maximize the useful image area.

alpha The free scaling parameter. If it is -1 , the functions performs some default scaling. Otherwise the parameter should be between 0 and 1. `alpha=0` means that the rectified images will be zoomed and shifted so that only valid pixels are visible (i.e. there will be no black areas after rectification). `alpha=1` means that the rectified image will be decimated and shifted so that all the pixels from the original images from the cameras are retained in the rectified images, i.e. no source image pixels are lost. Obviously, any intermediate value yields some intermediate result between those two extreme cases.

newImageSize The new image resolution after rectification. The same size should be passed to cv.InitUndistortRectifyMap, see the `stereo_calib.cpp` sample in OpenCV samples directory. By default, i.e. when (0,0) is passed, it is set to the original `imageSize`. Setting it to larger value can help you to preserve details in the original image, especially when there is big radial distortion.

roi1, roi2 The optional output rectangles inside the rectified images where all the pixels are valid. If `alpha=0`, the ROIs will cover the whole images, otherwise they likely be smaller, see the picture below

The function computes the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, that makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem. On input the function takes the matrices computed by cv. and on output it gives 2 rotation matrices and also 2 projection matrices in the new coordinates. The 2 cases are distinguished by the function are:

1. Horizontal stereo, when 1st and 2nd camera views are shifted relative to each other mainly along the x axis (with possible small vertical shift). Then in the rectified images the corresponding epipolar lines in left and right cameras will be horizontal and have the same y-coordinate. P1 and P2 will look as:

$$\texttt{P1} = \begin{bmatrix} f & 0 & cx_1 & 0 \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx_2 & T_x * f \\ 0 & f & cy & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_x is horizontal shift between the cameras and $cx_1 = cx_2$ if `CV_CALIB_ZERO_DISPARITY` is set.

2. Vertical stereo, when 1st and 2nd camera views are shifted relative to each other mainly in vertical direction (and probably a bit in the horizontal direction too). Then the epipolar lines in the rectified images will be vertical and have the same x coordinate. P2 and P2 will look as:

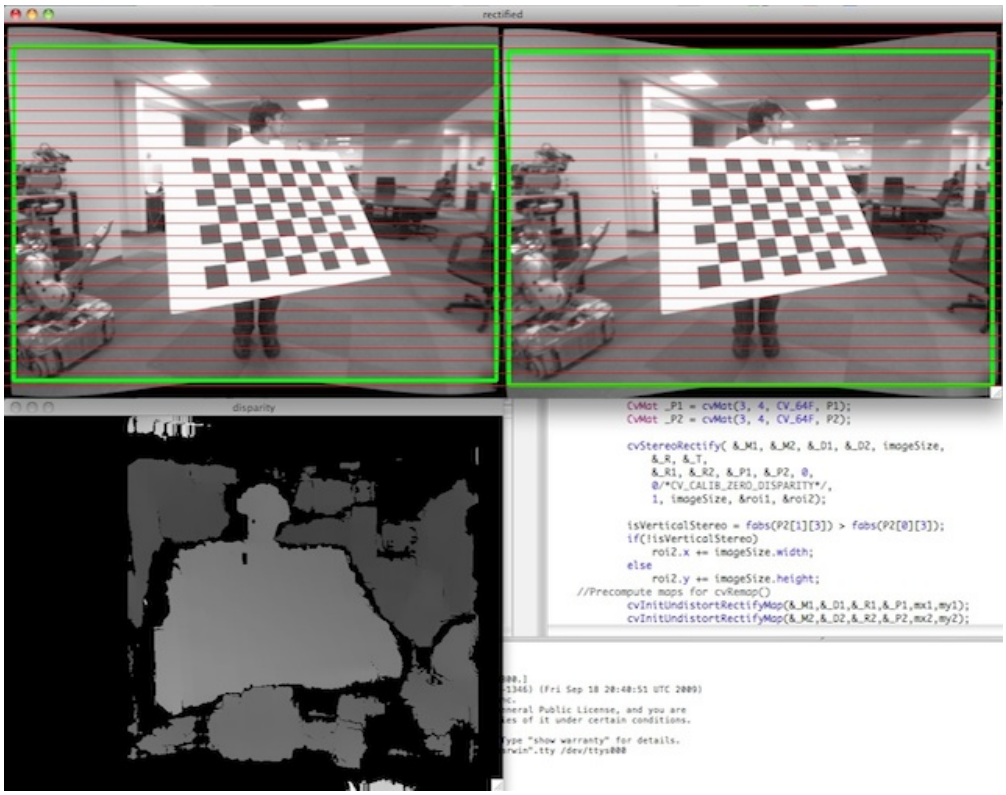
$$\texttt{P1} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\texttt{P2} = \begin{bmatrix} f & 0 & cx & 0 \\ 0 & f & cy_2 & T_y * f \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where T_y is vertical shift between the cameras and $cy_1 = cy_2$ if `CALIB_ZERO_DISPARITY` is set.

As you can see, the first 3 columns of `P1` and `P2` will effectively be the new "rectified" camera matrices. The matrices, together with `R1` and `R2`, can then be passed to cv.InitUndistortRectifyMap to initialize the rectification map for each camera.

Below is the screenshot from `stereo_calib.cpp` sample. Some red horizontal lines, as you can see, pass through the corresponding image regions, i.e. the images are well rectified (which is what most stereo correspondence algorithms rely on). The green rectangles are `roi1` and `roi2` - indeed, their interior are all valid pixels.



StereoRectifyUncalibrated

stereoRectifyUncalibrated Computes rectification transform for uncalibrated stereo camera.

```
StereoRectifyUncalibrated(points1,points2,F,imageSize,H1,H2,threshold=5)->
None
```

points1, points2 The 2 arrays of corresponding 2D points. The same formats as in cv.FindFundamentalMat are supported

F The input fundamental matrix. It can be computed from the same set of point pairs using cv.FindFundamentalMat.

imageSize Size of the image.

H1, H2 The output rectification homography matrices for the first and for the second images.

threshold The optional threshold used to filter out the outliers. If the parameter is greater than zero, then all the point pairs that do not comply the epipolar geometry well enough (that is, the points for which $|\texttt{points2[i]}^T * \texttt{F} * \texttt{points1[i]}| > \texttt{threshold}$) are rejected prior to computing the homographies. Otherwise all the points are considered inliers.

The function computes the rectification transformations without knowing intrinsic parameters of the cameras and their relative position in space, hence the suffix "Uncalibrated". Another related difference from cv.StereoRectify is that the function outputs not the rectification transformations in the object (3D) space, but the planar perspective transformations, encoded by the homography matrices `H1` and `H2`. The function implements the algorithm [?].

Note that while the algorithm does not need to know the intrinsic parameters of the cameras, it heavily depends on the epipolar geometry. Therefore, if the camera lenses have significant distortion, it would better be corrected before computing the fundamental matrix and calling this function. For example, distortion coefficients can be estimated for each head of stereo camera separately by using cv.CalibrateCamera2 and then the images can be corrected using cv.Undistort2, or just the point coordinates can be corrected with cv.UndistortPoints.

Undistort2

undistort Transforms an image to compensate for lens distortion.

```
Undistort2(src,dst,cameraMatrix,distCoeffs)-> None
```

src The input (distorted) image

dst The output (corrected) image; will have the same size and the same type as `src`

cameraMatrix The input camera matrix $A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The vector of distortion coefficients, $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$

The function transforms the image to compensate radial and tangential lens distortion.

The function is simply a combination of cv.InitUndistortRectifyMap (with unity `R`) and cv.Remap (with bilinear interpolation). See the former function for details of the transformation being performed.

Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with 0's (black color).

The particular subset of the source image that will be visible in the corrected image can be regulated by `newCameraMatrix`. You can use cv.GetOptimalNewCameraMatrix to compute the appropriate `newCameraMatrix`, depending on your requirements.

The camera matrix and the distortion parameters can be determined using cv.CalibrateCamera2. If the resolution of images is different from the used at the calibration stage, f_x, f_y, c_x and c_y need to be scaled accordingly, while the distortion coefficients remain the same.

UndistortPoints

undistortPoints Computes the ideal point coordinates from the observed point coordinates.

```
UndistortPoints(src,dst,cameraMatrix,distCoeffs,R=NULL,P=NULL)-> None
```

src The observed point coordinates, same format as `imagePoints` in cv.ProjectPoints2

dst The output ideal point coordinates, after undistortion and reverse perspective transformation , same format as `src` .

cameraMatrix The camera matrix $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$

distCoeffs The vector of distortion coefficients, $(k_1^{(j)}, k_2^{(j)}, p_1^{(j)}, p_2^{(j)}[, k_3^{(j)}])$

R The rectification transformation in object space (3x3 matrix). `R1` or `R2`, computed by cv. can be passed here. If the matrix is empty, the identity transformation is used

P The new camera matrix (3x3) or the new projection matrix (3x4). `P1` or `P2`, computed by cv. can be passed here. If the matrix is empty, the identity new camera matrix is used

The function is similar to cv.Undistort2 and cv.InitUndistortRectifyMap, but it operates on a sparse set of points instead of a raster image. Also the function does some kind of reverse transformation to cv.ProjectPoints2 (in the case of 3D object it will not reconstruct its 3D coordinates, of course; but for a planar object it will, up to a translation vector, if the proper `R` is specified).

```
// (u,v) is the input point, (u', v') is the output point
// camera_matrix=[fx 0 cx; 0 fy cy; 0 0 1]
// P=[fx' 0 cx' tx; 0 fy' cy' ty; 0 0 1 tz]
x" = (u - cx)/fx
y" = (v - cy)/fy
(x',y') = undistort(x",y",dist_coeffs)
```

```
[X,Y,W]T = R*[x' y' 1]T
x = X/W, y = Y/W
u' = x*fx' + cx'
v' = y*fy' + cy',
```

where undistort() is approximate iterative algorithm that estimates the normalized original point coordinates out of the normalized distorted point coordinates ("normalized" means that the coordinates do not depend on the camera matrix).

The function can be used both for a stereo camera head or for monocular camera (when R is None).

Chapter 14

cvaux. Extra Computer Vision Functionality

Chapter 15

highgui. High-level GUI and Media I/O

While OpenCV was designed for use in full-scale applications and can be used within functionally rich UI frameworks (such as Qt, WinForms or Cocoa) or without any UI at all, sometimes there is a need to try some functionality quickly and visualize the results. This is what the HighGUI module has been designed for.

It provides easy interface to:

- create and manipulate windows that can display images and ”remember” their content (no need to handle repaint events from OS)
- add trackbars to the windows, handle simple mouse events as well as keyboard commmands
- read and write images to/from disk or memory.
- read video from camera or file and write video to a file.

15.1 User Interface

---

cv.CreateTrackbar

Creates a trackbar and attaches it to the specified window

```
CreateTrackbar(trackbarName, windowName, value, count, onChange) ->
None
```

trackbarName Name of the created trackbar.

windowName Name of the window which will be used as a parent for created trackbar.

value Initial value for the slider position, between 0 and `count`.

count Maximal position of the slider. Minimal position is always 0.

onChange OpenCV calls `onChange` every time the slider changes position. OpenCV will call it as `func(x)` where `x` is the new position of the slider.

The function `cvCreateTrackbar` creates a trackbar (a.k.a. slider or range control) with the specified name and range, assigns a variable to be syncronized with trackbar position and specifies a callback function to be called on trackbar position change. The created trackbar is displayed on the top of the given window.

---

cv.DestroyAllWindows

Destroys all of the HighGUI windows.

```
DestroyAllWindows()-> None
```

The function `cvDestroyAllWindows` destroys all of the opened HighGUI windows.

---

cv.DestroyWindow

Destroys a window.

```
DestroyWindow(name)-> None
```

name Name of the window to be destroyed.

The function `cvDestroyWindow` destroys the window with the given name.

---

cv.GetTrackbarPos

Returns the trackbar position.

```
GetTrackbarPos(trackbarName,windowName)-> None
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of the trackbar.

The function cvGetTrackbarPos returns the current position of the specified trackbar.

---

cv.MoveWindow

Sets the position of the window.

```
MoveWindow(name,x,y)-> None
```

name Name of the window to be moved.

x New x coordinate of the top-left corner

y New y coordinate of the top-left corner

The function cvMoveWindow changes the position of the window.

---

cv.NamedWindow

Creates a window.

```
NamedWindow(name,flags=CV_WINDOW_AUTOSIZE)-> None
```

name Name of the window in the window caption that may be used as a window identifier.

flags Flags of the window. Currently the only supported flag is CV_WINDOW_AUTOSIZE. If this is set, window size is automatically adjusted to fit the displayed image (see ShowImage), and the user can not change the window size manually.

The function cvNamedWindow creates a window which can be used as a placeholder for images and trackbars. Created windows are referred to by their names.

If a window with the same name already exists, the function does nothing.

cv.ResizeWindow

Sets the window size.

```
ResizeWindow(name,width,height)-> None
```

name Name of the window to be resized.

width New width

height New height

The function `cvResizeWindow` changes the size of the window.

cv.SetMouseCallback

Assigns callback for mouse events.

```
SetMouseCallback(windowName, onMouse, param) -> None
```

windowName Name of the window.

onMouse Callable to be called every time a mouse event occurs in the specified window. This callable should have signature `Foo(event, x, y, flags, param)-> None` where `event` is one of `CV_EVENT_*`, `x` and `y` are the coordinates of the mouse pointer in image coordinates (not window coordinates), `flags` is a combination of `CV_EVENT_FLAG_*`, and `param` is a user-defined parameter passed to the `cvSetMouseCallback` function call.

param User-defined parameter to be passed to the callback function.

The function `cvSetMouseCallback` sets the callback function for mouse events occuring within the specified window.

The `event` parameter is one of:

CV_EVENT_MOUSEMOVE Mouse movement

CV_EVENT_LBUTTONDOWN Left button down

CV_EVENT_RBUTTONDOWN Right button down

CV_EVENT_MBUTTONDOWN Middle button down

CV_EVENT_LBUTTONUP Left button up

CV_EVENT_RBUTTONUP Right button up

CV_EVENT_MBUTTONUP Middle button up

CV_EVENT_LBUTTONDBLCLK Left button double click

CV_EVENT_RBUTTONDBLCLK Right button double click

CV_EVENT_MBUTTONDBLCLK Middle button double click

The `flags` parameter is a combination of :

CV_EVENT_FLAG_LBUTTON Left button pressed

CV_EVENT_FLAG_RBUTTON Right button pressed

CV_EVENT_FLAG_MBUTTON Middle button pressed

CV_EVENT_FLAG_CTRLKEY Control key pressed

CV_EVENT_FLAG_SHIFTKEY Shift key pressed

CV_EVENT_FLAG_ALTKEY Alt key pressed

---

cv.SetTrackbarPos

Sets the trackbar position.

```
SetTrackbarPos(trackbarName,windowName,pos)-> None
```

trackbarName Name of the trackbar.

windowName Name of the window which is the parent of trackbar.

pos New position.

The function `cvSetTrackbarPos` sets the position of the specified trackbar.

cv.ShowImage

Displays the image in the specified window

```
ShowImage(name,image)-> None
```

name Name of the window.

image Image to be shown.

The function cvShowImage displays the image in the specified window. If the window was created with the CV_WINDOW_AUTOSIZE flag then the image is shown with its original size, otherwise the image is scaled to fit in the window. The function may scale the image, depending on its depth:

- If the image is 8-bit unsigned, it is displayed as is.
- If the image is 16-bit unsigned or 32-bit integer, the pixels are divided by 256. That is, the value range [0,255*256] is mapped to [0,255].
- If the image is 32-bit floating-point, the pixel values are multiplied by 255. That is, the value range [0,1] is mapped to [0,255].

cv.WaitKey

Waits for a pressed key.

```
WaitKey(delay=0)-> int
```

delay Delay in milliseconds.

The function cvWaitKey waits for key event infinitely ($\texttt{delay} <= 0$) or for delay milliseconds. Returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing, unless HighGUI is used within some environment that takes care of event processing.

15.2 Reading and Writing Images and Video

cv.LoadImage

Loads an image from a file.

```
LoadImage(filename, iscolor=CV_LOAD_IMAGE_COLOR)->None
```

```
#define CV_LOAD_IMAGE_COLOR       1
#define CV_LOAD_IMAGE_GRAYSCALE   0
#define CV_LOAD_IMAGE_UNCHANGED  -1
```

filename Name of file to be loaded.

iscolor Specific color type of the loaded image: if > 0, the loaded image is forced to be a 3-channel color image; if 0, the loaded image is forced to be grayscale; if < 0, the loaded image will be loaded as is (note that in the current implementation the alpha channel, if any, is stripped from the output image, e.g. 4-channel RGBA image will be loaded as RGB).

The function `cvLoadImage` loads an image from the specified file and returns the pointer to the loaded image. Currently the following file formats are supported:

- Windows bitmaps - BMP, DIB
- JPEG files - JPEG, JPG, JPE
- Portable Network Graphics - PNG
- Portable image format - PBM, PGM, PPM
- Sun rasters - SR, RAS
- TIFF files - TIFF, TIF

cv.SaveImage

Saves an image to a specified file.

```
SaveImage(filename,image)-> None
```

filename Name of the file.

image Image to be saved.

The function cvSaveImage saves the image to the specified file. The image format is chosen based on the filename extension, see LoadImage . Only 8-bit single-channel or 3-channel (with 'BGR' channel order) images can be saved using this function. If the format, depth or channel order is different, use cvCvtScale and cvCvtColor to convert it before saving, or use universal cvSave to save the image to XML or YAML format.

CvCapture

Video capturing structure.

The structure CvCapture does not have a public interface and is used only as a parameter for video capturing functions.

---

cv.CaptureFromCAM

Initializes capturing a video from a camera.

```
CaptureFromCAM(index) -> CvCapture
```

index Index of the camera to be used. If there is only one camera or it does not matter what camera is used -1 may be passed.

The function cvCaptureFromCAM allocates and initializes the CvCapture structure for reading a video stream from the camera. Currently two camera interfaces can be used on Windows: Video for Windows (VFW) and Matrox Imaging Library (MIL); and two on Linux: V4L and FireWire (IEEE1394).

To release the structure, use ReleaseCapture .

cv.CaptureFromFile

Initializes capturing a video from a file.

```
CaptureFromFile(filename) -> CvCapture
```

filename Name of the video file.

The function `cvCaptureFromFile` allocates and initializes the CvCapture structure for reading the video stream from the specified file. Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect and how to prepare your video files.

After the allocated structure is not used any more it should be released by the ReleaseCapture function.

cv.GetCaptureProperty

Gets video capturing properties.

```
GetCaptureProperty(capture, property_id)->double
```

capture video capturing structure.

property_id Property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_FRAME_COUNT** Number of frames in the video file

CV_CAP_PROP_BRIGHTNESS Brightness of the image (only for cameras)

CV_CAP_PROP_CONTRAST Contrast of the image (only for cameras)

CV_CAP_PROP_SATURATION Saturation of the image (only for cameras)

CV_CAP_PROP_HUE Hue of the image (only for cameras)

The function `cvGetCaptureProperty` retrieves the specified property of the camera or video file.

---

cv.GrabFrame

Grabs the frame from a camera or file.

```
GrabFrame(capture) -> int
```

capture video capturing structure.

The function `cvGrabFrame` grabs the frame from a camera or file. The grabbed frame is stored internally. The purpose of this function is to grab the frame *quickly* so that syncronization can occur if it has to read from several cameras simultaneously. The grabbed frames are not exposed because they may be stored in a compressed format (as defined by the camera/driver). To retrieve the grabbed frame, RetrieveFrame should be used.

---

cv.QueryFrame

Grabs and returns a frame from a camera or file.

```
QueryFrame(capture) -> iplimage
```

capture video capturing structure.

The function `cvQueryFrame` grabs a frame from a camera or video file, decompresses it and returns it. This function is just a combination of GrabFrame and RetrieveFrame , but in one call. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cv.RetrieveFrame

Gets the image grabbed with cvGrabFrame.

```
RetrieveFrame(capture) -> iplimage
```

capture video capturing structure.

The function `cvRetrieveFrame` returns the pointer to the image grabbed with the GrabFrame function. The returned image should not be released or modified by the user. In the event of an error, the return value may be NULL.

cv.SetCaptureProperty

Sets video capturing properties.

```
SetCaptureProperty(capture, property_id,value)->None
```

capture video capturing structure.

property_id property identifier. Can be one of the following:

- **CV_CAP_PROP_POS_MSEC** Film current position in milliseconds or video capture timestamp
- **CV_CAP_PROP_POS_FRAMES** 0-based index of the frame to be decoded/captured next
- **CV_CAP_PROP_POS_AVI_RATIO** Relative position of the video file (0 - start of the film, 1 - end of the film)
- **CV_CAP_PROP_FRAME_WIDTH** Width of the frames in the video stream
- **CV_CAP_PROP_FRAME_HEIGHT** Height of the frames in the video stream
- **CV_CAP_PROP_FPS** Frame rate
- **CV_CAP_PROP_FOURCC** 4-character code of codec
- **CV_CAP_PROP_BRIGHTNESS** Brightness of the image (only for cameras)
- **CV_CAP_PROP_CONTRAST** Contrast of the image (only for cameras)
- **CV_CAP_PROP_SATURATION** Saturation of the image (only for cameras)
- **CV_CAP_PROP_HUE** Hue of the image (only for cameras)

value value of the property.

The function cvSetCaptureProperty sets the specified property of video capturing. Currently the function supports only video files: CV_CAP_PROP_POS_MSEC, CV_CAP_PROP_POS_FRAMES, CV_CAP_PROP_POS_AVI_RATIO.

NB This function currently does nothing when using the latest CVS download on linux with FFMPEG (the function contents are hidden if 0 is used and returned).

---

cv.CreateVideoWriter

Creates the video file writer.

```
CreateVideoWriter(filename, fourcc, fps, frame_size, is_color) ->
CvVideoWriter
```

filename Name of the output video file.

fourcc 4-character code of codec used to compress the frames. For example, CV_FOURCC('P','I','M','1') is a MPEG-1 codec, CV_FOURCC('M','J','P','G') is a motion-jpeg codec etc. Under Win32 it is possible to pass -1 in order to choose compression method and additional compression parameters from dialog. Under Win32 if 0 is passed while using an avi filename it will create a video writer that creates an uncompressed avi file.

fps Framerate of the created video stream.

frame_size Size of the video frames.

is_color If it is not zero, the encoder will expect and encode color frames, otherwise it will work with grayscale frames (the flag is currently supported on Windows only).

The function cvCreateVideoWriter creates the video writer structure.

Which codecs and file formats are supported depends on the back end library. On Windows HighGui uses Video for Windows (VfW), on Linux ffmpeg is used and on Mac OS X the back end is QuickTime. See VideoCodecs for some discussion on what to expect.

---

cv.WriteFrame

Writes a frame to a video file.

```
WriteFrame(writer, image)->int
```

writer Video writer structure

image The written frame

The function `cvWriteFrame` writes/appends one frame to a video file.

Chapter 16

ml. Machine Learning

The Machine Learning Library (MLL) is a set of classes and functions for statistical classification, regression and clustering of data.

Most of the classification and regression algorithms are implemented as C++ classes. As the algorithms have different seta of features (like the ability to handle missing measurements, or categorical input variables etc.), there is a little common ground between the classes. This common ground is defined by the class ‘CvStatModel‘ that all the other ML classes are derived from.

Bibliography

[1] M. Agrawal, K. Konolige, and M.R. Blas. Censure: Center surround extremas for realtime feature detection and matching. In *ECCV08*, pages IV: 102–115, 2008.

[2] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007.

[3] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *9th European Conference on Computer Vision*, Graz Austria, May 2006.

[4] Gunilla Borgefors. Distance transformations in digital images. *Comput. Vision Graph. Image Process.*, 34(3):344–371, 1986.

[5] Jean-Yves Bouguet. Pyramidal implementation of the lucas-kanade feature tracker, 2000.

[6] J.W. Davis and A.F. Bobick. The representation and recognition of action using temporal templates. In *CVPR97*, pages 928–934, 1997.

[7] J.W. Davis and G.R. Bradski. Motion segmentation and pose recognition with motion history gradients. In *WACV00*, pages 238–244, 2000.

[8] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Distance transforms of sampled functions. Technical report, Cornell Computing and Information Science, September 2004.

[9] R.I. Hartley. Theory and practice of projective rectification. *IJCV*, 35(2):115–127, November 1999.

[10] J. Matas, C. Galambos, and J.V. Kittler. Robust detection of lines using the progressive probabilistic hough transform. *CVIU*, 78(1):119–137, April 2000.

[11] F. Meyer. Color image segmentation. In *ICIP92*, page 303306, 1992.

[12] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Applications (VISSAPP'09)*, pages 331–340, 2009.

[13] J. Sklansky. Finding the convex hull of a simple polygon. *PRL*, 1:79–83, 1982.

[14] S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. *CVGIP*, 30(1):32–46, April 1985.

[15] C.H. Teh and R.T. Chin. On the detection of dominant points on digital curve. *PAMI*, 11(8):859–872, August 1989.

[16] Alexandru Telea. An image inpainting technique based on the fast marching method. *Journal of Graphics, GPU, and Game Tools*, 9(1):23–34, 2004.

[17] C. Tomasi and J. Shi. Good features to track. In *CVPR94*, pages 593–600, 1994.

[18] Greg Welch and Gary Bishop. An introduction to the kalman filter, 1995.

[19] H. K. Yuen, J. Princen, J. Illingworth, and J. Kittler. Comparative study of hough transform methods for circle finding. *Image Vision Comput.*, 8(1):71–77, 1990.

Index